ISOLAMENTO DE TRÁFEGO EFICIENTE PARA DATACENTERS VIRTUALIZADOS

HENRIQUE DA SILVA RODRIGUES

ISOLAMENTO DE TRÁFEGO EFICIENTE PARA DATACENTERS VIRTUALIZADOS

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Ciências Exatas da Universidade Federal de Minas Gerais como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

Orientador: Dorgival Olavo Guedes Neto

Belo Horizonte

Março de 2011

Rodrigues, Henrique da Silva
R696i Isolamento de Tráfego Eficiente para Datacenters Virtualizados / Henrique da Silva Rodrigues. — Belo Horizonte, 2011
xiv, 108 f. : il. ; 29cm

Dissertação (mestrado) — Universidade Federal de Minas Gerais
Orientador: Dorgival Olavo Guedes Neto

1. Redes. 2. Sistemas Distribuídos. 3. Qualidade de Serviço. 4. Computação em Núvem. 5. Virtualização. I. Título.

CDU 519.6*22(043)

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

FOLHA DE APROVAÇÃO

Isolamento de tráfego eficiente para datacenters virtualizados

HENRIQUE DA SILVA RODRIGUES

Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores:

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

DR. JOSÉ RENATO SANTOS
Hewlett-Packard Laboratories

PROF. JOSÉ MARCOS SILVA NOGUEIRA
Departamento de Ciência da Computação - UFMG

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

Belo Horizonte, 28 de julho de 2011.

Resumo

Datacenters que oferecem infraestrutura computacional no modelo de computação em nuvem abrigam aplicações de diversos clientes em um ambiente compartilhado. Essas aplicações são executadas por uma ou mais máquinas virtuais (VMs) que compartilham os servidores do datacenter. Com a tendência de terceirização de infraestrutura, motivada pela popularização do modelo de *computação como serviço*, cresce o número de clientes em um mesmo datacenter. Enquanto isso, provedores de infraestrutura buscam um melhor aproveitamento dos recursos consolidando um grande número de VMs em um único servidor. Esses fatores aumentam a interação de diferentes clientes no consumo de recursos. Nesse ambiente é importante que o provedor garanta o isolamento no consumo de recursos de diferentes clientes, evitando violações nas regras contratuais que especificam os recursos contratados por cada cliente.

Embora as atuais tecnologias de virtualização ofereçam boas soluções para o isolamento de máquinas virtuais em termos de memória e processamento, o mesmo não ocorre para os recursos de rede. Essas tecnologias são capazes de controlar apenas a transmissão de dados e normalmente dependem da cooperação das VMs para controlar a recepção de dados de forma justa. Isso torna o isolamento do consumo de recursos de rede vulnerável à padrões de tráfego "egoístas", que não fazem qualquer esforço para manter uma divisão justa dos recursos de rede.

O Gatekeeper é uma iniciativa para resolver esse problema. Esse sistema detecta violações nas garantias dos recursos adquiridos pelos clientes e coordena a ação de limitadores de envio de tráfego associados à cada VM para acabar com as violações. Esse controle é feito de forma distribuída com a participação de cada servidor do datacenter. A arquitetura e os algoritmos do Gatekeeper, no entanto, possuem algumas limitações que resultam em um controle de tráfego ineficiente em determinados cenários. Esse trabalho propõe uma nova versão para o sistema, denominada Gatekeeper-ng, que resolve esses problemas e oferece um controle de tráfego mais preciso e mais eficiente.

Palavras-chave: Redes, Sistemas Distribuídos, Qualidade de Serviço, Virtualização.

Abstract

A typical cloud-based datacenter hosts multiple services owned by various *tenants* in a shared facility. Each service can consist of a collection of one or more virtual machines (VM) placed on one or more physical machines. With the trend of outsourcing the infrastructure, motivated by the wide spread of the utility computing model, the number of tenants sharing the computational resources of a single datacenter is also expected to grow. Meanwhile, infrastructure providers are adopting server consolidation techniques to maximize the resource utilization by placing even more VMs on each server. These factors increase the interactions on resource consumption among different tenants. Thus, such shared environments will have a strong need for improved mechanisms to enforce performance isolation for tenants that share datacenter resources.

While existing virtualization mechanisms provide good support for allocating CPU and memory resources, only rudimentary support is currently available to manage the use of datacenter network I/O resources. Today's virtualization solutions can enforce resource consumption in the transmit direction, but the reception still depends on servers' cooperation. Therefore, these traffic isolation solutions are vulnerable to "selfish" tenants, which doesn't cooperate to share the network resources in a fairly manner.

The Gatekeeper system is an initiative to solve this problem. The system detects resource consumption violations according to traffic guarantees specified by the datacenter operator and enforces bandwitdh allocations for each VM, for both egress and ingress traffic, on each server. However, the algorithms and the architecture of Gatekeeper have limitations that lead to poor performance isolation in some scenarios. This work proposes a new version of the system, called Gatekeeper-ng, that solves these problems.

Keywords: Computer Networks, Distributed Systems, Quality of Service, Virtualization.

Lista de Figuras

Sumário

Capítulo 1

Introdução

A computação em nuvem (*Cloud Computing*) é um novo modelo de provisão de recursos computacionais que está cada vez mais presente tanto na academia quanto em aplicações comerciais [69]. Esse emergente modelo afeta diretamente a tradicional infraestrutura de TI, seus serviços e aplicações. A facilidade relacionada ao acesso e à gestão dos recursos computacionais são alguns dos principais fatores que contribuíram para a sua popularização. Os recursos providos com base no modelo podem ser alocados ou liberados a qualquer instante, de acordo com a necessidade de cada usuário. Além disso, a transparência com que os recursos podem ser gerenciados pelos usuários livra-os da obrigação de lidar com a difícil tarefa de gerir uma infraestrutura computacional complexa. Atualmente, a computação em nuvem é adotada por algumas das maiores empresas de tecnologia, empresas de porte médio, várias *startups* e também grupos de pesquisa, atraídos pelas vantagens desse novo modelo de provisão de recursos [5, 28].

Na computação em nuvem, a *nuvem* é uma referência aos recursos computacionais localizados em diversos *datacenters* [1] (DCs) gerenciados por provedores de infraestrutura computacional. As peculiaridades envolvidas no modo com que esses recursos são disponibilizados aos usuários dão forma ao modelo, sendo suas principais características a facilidade e a transparência no acesso aos recursos, a infinidade de recursos disponíveis e o modelo de preços flexível. O usuário geralmente não possui informações sobre a infraestrutura na qual os recursos estão localizados, mas consegue acessá-los de forma transparente. O provedor de infraestrutura, por sua vez, se compromete a disponibilizar uma grande quantidade de recursos, que os usuários podem decidir por utilizar ou não, dependendo de sua necessidade. O custo flexível é proveniente da possibilidade de contratação (e liberação) de recursos sob demanda, dispensando a

[1] O termo *datacenter* vem sendo usado sem tradução na literatura em português e, por esse motivo, será adotado no restante deste trabalho sem ênfase em itálico.

necessidade de contratos fixos de longo prazo. O cliente paga apenas pela quantia de recursos utilizada, sendo a cobrança de consumo por hora o modelo de comercialização mais comum atualmente.

A tecnologia chave para a implementação da computação em nuvem é a Virtualização [7, 39, 59, 63]. No uso mais comum dessa tecnologia, Monitores de Máquinas Virtuais (*Virtual Machine Monitors — VMMs*) são softwares que gerenciam o acesso aos recursos do hardware de uma máquina física de forma a criar ambientes operacionais abstratos, chamados de máquinas virtuais (*Virtual Machines — VMs*). Uma máquina virtual provê ao usuário as mesmas funcionalidades de uma maquina física, sendo capaz de executar softwares originalmente desenvolvidos para máquinas físicas sem quaisquer modificações. No contexto da computação em nuvem, a virtualização possibilita que um datacenter hospede múltiplos serviços de clientes distintos em um ambiente compartilhado. Os serviços são frequentemente consolidados em máquinas físicas utilizando máquinas virtuais. Na visão do provedor de recursos, uma VM é uma caixa preta que apresenta uma certa demanda por recursos computacionais. Essa visão facilita a gerência dos recursos do datacenter por parte do provedor, que deve decidir como as máquinas virtuais deverão ser alocadas em suas máquinas físicas de acordo com as demandas de cada uma e dos requisitos definidos pelo usuário.

Internamente, uma nuvem acessível a um usuário é composta por um ou mais datacenters que comercializam recursos computacionais seguindo o modelo da computação em nuvem. Cada serviço hospedado em um datacenter virtualizado pode consistir em uma coleção de uma ou mais VMs alocadas em uma ou mais máquinas físicas. Esses ambientes são compartilhados por diversos clientes e abrigam uma grande variedade de serviços, como soluções para Correio Eletrônico e Armazenamento de Dados, Servidores de Páginas Web, de Jogos e de Mensagens Instantâneas [8]. Além desses, datacenters compartilhados também podem hospedar aplicações intensivas em dados como sistemas de indexação de páginas Web e/ou mineração de dados.

Com o constante crescimento da computação em nuvem, existe uma tendência de que tais serviços pertencerão a uma grande quantidade de clientes mutuamente não confiáveis e exibirão demandas dinâmicas variadas sobre os recursos do datacenter [32]. Essa característica de cada cliente apresentar uma reserva de recursos por tempo limitado levou os autores de língua inglesa a se referir a eles como "inquilinos" do datacenter (*tenants*), nomenclatura que vem também sendo adotada (com ou sem tradução) em português.

1.1 Motivação

A grande diversidade e dinamicidade no consumo de recursos em uma nuvem faz com que esses novos ambientes necessitem de bons mecanismos para manter o isolamento de desempenho entre os serviços pertencentes a diferentes usuários. Para fins de garantir a qualidade do serviço oferecido a cada um dos seus clientes, os recursos disponíveis para cada máquina virtual são descritos por Acordos de Níveis de Serviço (*Service Level Agreements — SLAs*), firmados entre o provedor de recursos e os clientes de um datacenter. Características como poder de processamento e capacidade de armazenamento, por exemplo, são alguns dos itens que compõem um SLA. Cabe ao provedor de recursos dividir os recursos físicos entre as máquinas virtuais e garantir a disponibilidade dos mesmos. O desafio nesse cenário, que conta com uma infraestrutura compartilhada e demandas variáveis, é garantir a disponibilidade dos recursos contratados sempre que houver demanda para seu consumo, constantemente observando as garantias de cada cliente — determinadas pelos SLAs.

Essa necessidade se torna ainda mais evidente frente à consolidação de servidores (*Server Consolidation*), uma estratégia comum para reduzir os custos de um datacenter [69]. O objetivo é aumentar a eficiência do datacenter reduzindo o consumo desnecessário de recursos da infraestrutura e, consequentemente, o custo de operação (o consumo energético de um datacenter de 50 mil servidores pode ser maior que 9 milhões de dólares por ano, segundo a estimativa de [19], sendo a infraestrutura e resfriamento responsáveis por 59% e 33% desse valor, respectivamente). Utilizando migrações de máquinas virtuais, que é um recurso comum atualmente, Sistemas de Gerenciamento de Recursos são capazes de alterar a alocação das VMs em tempo de execução, sem a necessidade de interromper os serviços oferecidos por elas. Essa estratégia, quando aplicada em um datacenter com recursos compartilhados, aumenta a interação do consumo de recursos entre diferentes clientes, pois suas máquinas virtuais são "espremidas" em um conjunto menor de máquinas físicas. Com isso, máquinas físicas — assim como dispositivos da malha de rede — não utilizadas podem ser desativadas temporariamente.

Embora as atuais tecnologias de virtualização ofereçam boas soluções para o isolamento das máquinas virtuais em termos de memória e processamento (CPU), operações de entrada e saída (E/S) relacionadas à rede são ainda um problema (o VMWare ESX Server 3, por exemplo, consegue garantir o consumo máximo e médio da largura de banda de cada VM, mas a solução se aplica apenas para a transmissão de dados: não há garantias para a recepção de dados [62]).

A malha de rede utilizada em um datacenter é, conceitualmente, um *recurso*

distribuído que é compartilhado entre diversos fluxos de tráfego utilizando comutação por pacotes [6, 45]. Por outro lado, CPU e memória são *recursos locais* e, portanto, mais fáceis de serem controlados. Para demonstrar a diferença entre esses dois tipos de recursos, considere que o controle dos recursos de rede seja feito da mesma forma que o controle de CPU e memória: limitando, em cada máquina física, a utilização das VMs à uma fração da capacidade da interface de rede. Isso seria equivalente a limitar uma VM à utilização de um único núcleo da CPU ou a uma porção limitada da memória. Nesse cenário, um único cliente com um grande número de VMs poderia acumular os limites de rede de cada uma de suas VMs e utilizar toda a capacidade de um dos enlaces da rede (ou a capacidade total de recepção de uma única interface de rede), reduzindo a qualidade do serviço de rede oferecido a algum outro cliente.

O controle eficiente dos recursos de rede também será crucial para suportar a crescente diversidade de serviços e aplicações que injetam demandas massivas na rede de datacenters compartilhados. Aplicações intensivas em dados que fazem uso de sistemas de armazenamento distribuídos ou ambientes de programação altamente escaláveis, como o MapReduce [15], podem ser intensivas no uso da rede. Quando executadas em um datacenter virtualizado essas aplicações podem, mesmo que não intencionalmente, gerar padrões de tráfego que prejudiquem o desempenho de outras aplicações hospedadas no mesmo ambiente compartilhado. Um exemplo é um volume massivo de conexões TCP de curta duração operando em paralelo: por sua curta duração, a maior parte do tráfego é enviado durante a fase *slow start* do protocolo TCP, quando o controle de congestionamento não está ativo. Além disso, tráfego que não possui controle de congestionamento, como por exemplo tráfego UDP ou implementações maliciosas do protocolo TCP, podem ser utilizados em um datacenter (já que a pilha de protocolos de rede se encontra em cada VM, fora do controle do operador do datacenter). Esses dois tipos de tráfego foram utilizados, por exemplo, para deflagrar um ataque de negação de serviço contra o Bitbucket, um sistema de controle de versões baseado na Web que possui mais de 60 mil usuários [9]. Na época do ataque, o Bitbucket utilizava máquinas virtuais providas pela Amazon EC2 [18] para hospedar os seus serviços; como não havia um mecanismo de controle de tráfego eficiente na rede da Amazon, o serviço foi afetado diretamente.

1.2 Objetivos

Dada a necessidade de um mecanismo de controle de tráfego capaz de prover isolamento de desempenho entre os diversos clientes de uma nuvem pública, como parte de uma

dissertação anterior, foi desenvolvido o Gatekeeper [54, 55]. Composto por agentes de controle de tráfego instalados em cada máquina física do datacenter, o Gatekeeper gerencia as limitações de tráfego de cada máquina virtual utilizando um protocolo de controle distribuído.

O objetivo deste trabalho de mestrado foi estudar e propor melhorias para o Gatekeeper. Mais especificamente, este trabalho se concentra em aprimorar as garantias de tráfego providas por aquele sistema, aumentando a precisão do controle de rede oferecido pelo mesmo. A primeira implementação do Gatekeeper deixou oportunidades para melhorias, tanto relacionadas à arquitetura do sistema quanto aos algoritmos de controle de tráfego, que exploravam as informações de tráfego providas pelo sistema operacional apenas superficialmente.

Além das melhorias nos algoritmos, outro objetivo do trabalho foi a reimplementação do sistema utilizando recursos mais modernos. Em particular, a nova versão foi implementada utilizando-se o Open vSwitch, uma nova arquitetura de *switches* virtuais para servidores virtualizados.

1.3 Contribuições

A contribuição desse trabalho foi o desenvolvimento do Gatekeeper-ng, uma nova versão do Gatekeeper com um controle de tráfego mais eficaz. A nova implementação conta com uma nova arquitetura e novos algoritmos de alocação de tráfego mais eficientes e que apresentam maior escalabilidade quando comparados aos algoritmos propostos inicialmente.

A nova arquitetura possibilita o uso de informações detalhadas a respeito do tráfego de rede trocado entre as VMs. Essa informação é utilizada pelos algoritmos de alocação de tráfego para determinar limites de banda ótimos para cada máquina virtual, melhorando assim a precisão do controle de tráfego oferecido pelo sistema. Além disso, o Gatekeeper-ng impõe menor sobrecarga às máquinas físicas, pois parte dele é implementado dentro do núcleo do sistema operacional.

Foi conduzida uma avaliação experimental comparativa entre os dois sistemas avaliando, entre outros aspectos, a escalabilidade, a eficiência e a precisão dos dois algoritmos de controle de tráfego. A partir da experimentação realizada foi possível mostrar que o Gatekeeper-ng é uma solução mais escalável e que oferece um melhor controle de tráfego que o Gatekeeper, enquanto que ao mesmo tempo impõe uma menor sobrecarga à CPU.

Uma outra contribuição deste trabalho foi a integração do Gatekeeper ao arca-

bouço *Open vSwitch* [46], um elemento de chaveamento para ambientes virtualizados baseado na arquitetura *OpenFlow* [37]. Essa integração, além de simplificar a implementação da solução, torna o Gatekeeper-ng disponível no contexto de um arcabouço que vem se tornando muito popular na área de virtualização. *Open vSwitch* [46] e *OpenFlow* [37] serão descritos em mais detalhes na seção 2.5.2.

Um outro ponto de destaque do trabalho foi a colaboração com pesquisadores do HP Labs, Palo Alto, que cogitam a possibilidade de integrar o Gatekeeper-ng a uma solução comercial da empresa.

1.4 Estrutura do Texto

O restante desta dissertação está organizado da seguinte forma: o capítulo seguinte introduz os conceitos básicos associados a esse trabalho, o capítulo 3 descreve a implementação do Gatekeeper-ng e o capítulo 4 apresenta uma avaliação do seu desempenho.

Em particular, o capítulo 2 introduz os princípios de virtualização e controle de tráfego em redes, apresenta a estrutura de redes de datacenters e descreve os princípios gerais da implementação original do Gatekeeper e do arcabouço *Open vSwitch*, que foi utilizado no Gatekeeper-ng. O capítulo é concluído com uma discussão de outros trabalhos relacionados ao problema de garantias de tráfego em datacenters.

O capítulo 3 apresenta a arquitetura proposta para a nova versão do Gatekeeper, detalha os algoritmos envolvidos no processo de controle de alocação de banda e discute os detalhes de implementação, incluindo as alterações necessárias no arcabouço *Open vSwitch* para viabilizar a operação do Gatekeeper-ng. Essa discussão ilustra como a nova arquitetura oferece ganhos na redução de complexidade e aumento da capacidade de controle do novo sistema.

O sistema implementado apresenta um desempenho significativamente superior ao Gatekeeper original. Os resultados de diversos testes comparativos compõem o capítulo 4, considerando também outras soluções para controle de tráfego usualmente adotadas.

Conclusões e uma discussão dos trabalhos futuros completam a parte principal do texto, no capítulo 5. Além disso, os apêndices fornecem mais detalhes sobre aspectos de programação no *kernel* e sobre a implementação das diversas partes do sistema.

Capítulo 2

Conceitos Relacionados

Esse capítulo apresenta os conceitos científicos e tecnológicos envolvidos no desenvolvimento deste trabalho. Os principais tópicos abordados são a virtualização de recursos computacionais, o problema de controle de tráfego em redes de computadores, o modelo de redes adotado em datacenters, a alocação de recursos de rede de forma justa como implementada originalmente no Gatekeeper e o arcabouço *Open vSwitch*, utilizado na nova implementação. O capítulo conclui com uma discussão de outros trabalhos relacionados.

2.1 Virtualização

Apesar da recente expansão do uso de virtualização, o conceito de sistemas virtualizados não é recente. A idealização das primeiras máquinas virtuais (*Virtual Machines, VMs*) datam da década de 60, quando a IBM começou a utilizá-las para prover acesso concorrente e interativo aos seus computadores *mainframes* [43]. Naquela época, o custo de um *mainframe* era muito alto e a virtualização passou a ser considerada uma forma interessante de compartilhar o hardware entre diversos usuários melhorando o aproveitamento dos recursos gastos com esses equipamentos. Em um ambiente virtualizado, os usuários têm a impressão de que suas máquinas virtuais são instâncias de uma máquina física, não existindo diferenças relacionadas ao acesso dos recursos de hardware. Cada VM era uma cópia quase exata do sistema *mainframe* e os seus usuários eram capazes de desenvolver e executar aplicações sem ter que se preocupar com falhas que poderiam prejudicar o funcionamento do sistema como um todo, prejudicando também as tarefas de outros usuários que estavam sendo executadas no mesmo hardware. A virtualização tornou-se uma forma ideal de diminuir o custo da aquisição

de novos equipamentos e também melhorar a produtividade com um novo modelo de compartilhamento de recursos.

À medida que o custo da aquisição de hardware caiu com o passar do tempo, as vantagens da virtualização também começaram a diminuir. Isso fez com que o uso dessa tecnologia fosse quase extinto. Porém, com o aparecimento de diferentes produtores de hardware e suas diferentes arquiteturas na década de 90, a virtualização passou a ser útil novamente. Nesse novo cenário, ela possibilitava que aplicações projetadas para diferentes arquiteturas fossem executadas em uma mesma máquina física sem a necessidade de recompilar ou de modificar o código-fonte das aplicações. Desde então a adoção da virtualização voltou a crescer novamente. Nos dias de hoje, com o surgimento do conceito recente de computação em nuvem, a virtualização tem se mostrado cada vez mais presente tanto na academia quanto em aplicações comerciais.

Algumas das características e/ou vantagens relacionadas ao uso da tecnologia podem ser resumidos da seguinte forma:

- **Sandboxing:** Máquinas virtuais podem ser vistas como unidades de recursos capazes de prover ambientes seguros e isolados (*sandboxes*) para a execução de aplicações mutuamente não confiáveis — que podem danificar o sistema como um todo.

- **Melhor utilização de recursos/Consolidação de servidores:** De modo geral, sistemas de computação não utilizam todos os recursos disponíveis durante todo o tempo, exibindo uma oscilação na taxa de utilização de recursos [4, 11, 51, 60, 65]. A distribuição dos recursos de uma máquina física entre várias máquinas virtuais pode diminuir o desperdício de recursos que estão ociosos em grande parte do tempo.

- **Facilidade e Segurança no Teste de Software:** Acionadores de dispositivos ou de módulos de sistemas operacionais, que acessam diretamente os recursos de hardware, podem comprometer o sistema como um todo no caso de conterem um erro de programação. Desenvolvedores desse tipo de software podem utilizar máquinas virtuais para testar esses componentes ao invés de executá-los diretamente no hardware alvo. Isso facilita e agiliza o desenvolvimento desses componentes de software, pois mesmo que contenham um erro grave, os outros sistemas que executam na mesma máquina física ainda continuam em execução, podendo auxilar na identificação e depuração do problema.

- **Migração de VMs e Facilidade de Manutenção:** Como uma VM é basicamente um módulo de software executando em uma máquina física, é possível

utilizar técnicas de migração de processos para mover VMs de uma máquina para outra. Essa mobilidade possibilita que o operador do datacenter migre as máquinas virtuais em execução de uma máquina física para outra, possibilitando o desligamento da máquina física de origem para fins de manutenção, por exemplo. Além da migração, também é possível fazer *checkpoints* de máquinas virtuais, que podem ser utilizados como um mecanismo de tolerância a falhas.

- **Pacotes de Software completos (*Virtual Appliances*):** Ao distribuir um pacote de software, uma das práticas que está se tornando comum é o uso de *Virtual Appliances*: uma máquina virtual com um sistemas operacional que atende exatamente (e muitas vezes apenas) as necessidades do software distribuído. Com isso é possível evitar problemas de dependências de módulos e bibliotecas inexistentes do ambiente de destino[1] e ter um pacote de software pronto para a execução imediata em uma plataforma de virtualização específica. Isso também reduz o tempo de instalação do software, que normalmente já vem pré-instalado e pré-configurado na máquina virtual disponibilizada.

- **Múltiplos Sistemas Operacionais:** Com a virtualização é possível executar diferentes sistemas operacionais ao mesmo tempo utilizando uma mesma máquina física. Isso pode ser de maior utilizada para usuários finais, que podem precisar de diferentes sistemas para diferentes tarefas, ou para administradores de rede, que podem agregar serviços de diferentes sistemas em um mesmo hardware.

Devido às vantagens da virtualização, estima-se que a sua adoção continue a crescer e que em menos de 5 anos, 63% dos processadores desenvolvidos com base na arquitetura x86 executarão sistemas virtualizados [46]. Nos dias de hoje, o uso da virtualização é mais comum em servidores, porém, existem projetos que propõem a utilização das vantagens da virtualização em estações de trabalho, como por exemplo o *XenDesktop* da Citrix [13].

O crescimento das aplicações da virtualização em diversas áreas da ciência da computação faz com que a definição do termo se torne cada vez mais abrangente. Uma das definições que é capaz de expressar grande parte da aplicabilidade e delinear os objetivos relacionados à virtualização é a feita por [43]: "Virtualização é a tecnologia que combina ou divide recursos computacionais provendo a abstração de um ou mais ambientes operacionais, utilizando para isso técnicas de particionamento ou agregação de hardware e software, simulação e/ou emulação de máquinas físcias de diferentes

[1] Problema chamado de *Dependency Hell* em ambientes Linux e *DLL Hell* em ambientes Windows.

arquiteturas, compartilhamento de capacidade de processamento baseado em tempo (*time-sharing*), dentre outras."

2.1.1 Modos de Implementação

É comum fazer a associação do conceito de virtualização com o uso de máquinas virtuais. No entanto, existem diversas formas de prover virtualização e cada uma delas provê diferentes níveis de flexibilidade e de aproveitamento de recursos computacionais.

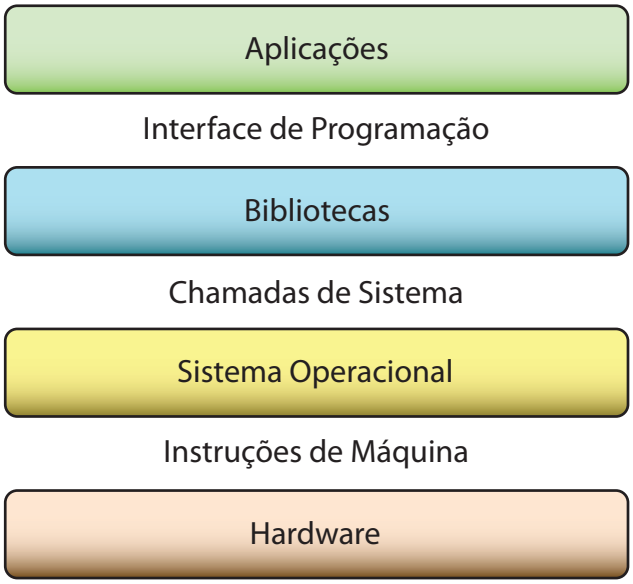


Figura 2.1. Visão geral da virtualização e de onde ela pode ser implementada

A figura 2.1 apresenta uma visão geral de onde a virtualização pode ser implementada adotando como base a divisão das camadas de software normalmente presentes em um servidor convencional. Algumas das opções são:

- **Virtualização no nível das instruções de hardware:** Nesse caso um ambiente virtualizado é provido através da tradução das instruções de máquina de uma aplicação para as instruções uma arquitetura diferente. O processo é feito inteiramente em software e possibilita que uma aplicação desenvolvida para uma arquitetura seja executada em outros processadores.

  Como o conjunto de instruções que compõem a aplicação devem ser interpretados e traduzidos para o conjunto de instruções da arquitetura alvo, esse tipo de virtualização também é frequentemente chamado de emulação.

- **Virtualização na camada de abstração de hardware (*Hardware Abstraction Layer — HAL*):** Essa técnica provê um ambiente virtualizado que

combina as características da máquina física com as funcionalidades da emulação do conjunto de instruções. É a técnica de virtualização mais utilizada atualmente e também a que está mais relacionada com grande parte das vantagens listadas na seção anterior.

Na virtualização HAL uma camada de software comumente chamada de monitor de máquinas virtuais (*Virtual Machine Monitor, VMM*) ou *hypervisor* gerencia o acesso de múltiplas máquinas virtuais ao hardware, oferecendo funcionalidades semelhantes às da máquina física para cada uma das VMs. A principal característica dessa técnica é o sacrifício da flexibilidade em termos de arquitetura computacional em favor do ganho de desempenho A ideia é unificar a arquitetura da máquina física e das máquinas virtuais de forma a executar as instruções dos aplicativos das máquinas virtuais diretamente no hardware, sem que haja perda de desempenho. A exceção são as instruções privilegiadas, como as de acesso aos recursos de hardware específicos, que devem ser tratadas pelo VMM para garantir a segurança entre as várias VMs que compartilham a mesma máquina física. O VMM também é responsável por realizar o escalonamento de máquinas virtuais, decidindo qual a parcela do tempo do processador deve ser dado a cada uma delas.

- **Virtualização a nível de API:** Aplicações normalmente utilizam um conjunto de bibliotecas de software para interagir com o sistema operacional. Essa prática facilita o seu desenvolvimento, pois bibliotecas de software normalmente contêm grande parte das tarefas básicas de programação já implementadas, como tratamento de erros e tratamento de exceções. Além disso, bibliotecas de software quase sempre possuem uma especificação muito bem definida. Isso faz com que seja possível reimplementar bibliotecas inteiras possibilitando executar aplicações em sistemas distintos. Essa estratégia provê um ambiente virtualizado no nível de interfaces de programação (API). Um sistema de virtualização muito popular atualmente que utiliza uma estratégia semelhante para possibilitar a execução de aplicações Windows em sistemas Linux é o Wine [67].

- **Máquina virtual completa no nível de aplicação:** Uma outra forma de prover um ambiente virtual é através da implementação de um sistema todo virtualizado para a execução de aplicações específicas, como é feito com a máquina virtual da linguagem Java, a JVM.

Como o objetivo desse trabalho é controlar a divisão dos recursos de rede entre várias VMs de forma eficiente, é desejável que o modo de implementação da virtua-

lização adotado para a realização do trabalho seja capaz de controlar os recursos do hardware diretamente. A implementação que mais se aproxima desse requisito é a realizada no nível de HAL. A próxima seção apresenta a arquitetura e os conceitos básicos da solução de virtualização a nível de HAL adotada para o desenvolvimento deste trabalho.

2.1.2 Xen

O Xen é um monitor de máquinas virtuais de código aberto que implementa virtualização no nível da camada de abstração de hardware. As primeiras versões do sistema foram desenvolvidas por pesquisadores da Universidade de Cambridge e ele é atualmente mantido pela Comunidade Xen. É um dos sistemas de virtualização mais utilizados por plataformas de computação em nuvem, inclusive pela popular *Amazon Elastic Compute Cloud (EC2)* [18]. Além disso, por ser uma ferramenta de código aberto, o sistema também é muito utilizado por pesquisadores, que podem implementar e experimentar novas soluções para sistemas virtualizados utilizando o Xen. Esse também é um dos motivo pelo qual o Xen foi a plataforma de virtualização adotada para o desenvolvimento desse trabalho.

O sistema é composto por dois componentes principais, o *hypervisor* (VMM) e um ambiente virtualizado que tem permissões especiais, chamado de Domínio Privilegiado ou Domínio0 (Dom0). A figura 2.2 apresenta uma visão geral dos componentes de uma máquina virtualizada com Xen. O hypervisor é responsável por gerenciar o acesso ao hardware e controlar o consumo de recursos de cada máquina virtual. O Dom0 é uma máquina virtual especial utilizada para gerenciar a atribuição de recursos de cada VM, além de ser responsável por criar e remover outras VMs.

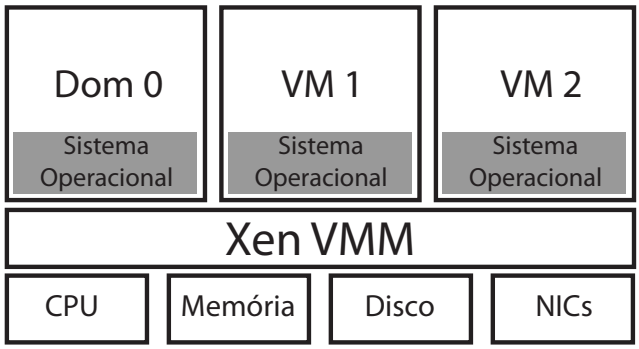


Figura 2.2. Visão geral da virtualização oferecida pelo Xen

O Xen oferece dois diferentes modos de operação para a execução de máquinas virtuais que são implementados no nível de HAL:

- **Virtualização Completa:** Nesse modo o sistema executado no ambiente virtualizado não precisa de quaisquer alterações e o VMM cria a ilusão de que o sistema está sendo executado em uma máquina física comum. No entanto é necessário que todas as instruções privilegiadas (que tentam acessar os recursos do hardware indevidamente) sejam tratadas pelo VMM. Isso requer o uso de métodos de análise dinâmica de código para reescrever as instruções que devem ser tratadas pelo VMM em tempo de execução. O custo computacional para realizar essas operações pode causar um impacto significativo no desempenho dos sistemas virtualizados, mesmo utilizando soluções como *caching* de trechos de código frequentes. Por esse motivo, a virtualização completa não é o principal modo de operação de máquinas virtuais de um ambiente Xen.

- **Para-virtualização:** Um dos diferenciais do Xen é a sua implementação do conceito de para-virtualização[2], que possibilita a execução de máquinas virtuais com desempenho muito semelhante à execução nativa, isto é, diretamente no hardware e sem o uso da virtualização.

  Esse modo de operação do Xen é mais intrusivo que a virtualização completa, exigindo modificações no sistema operacional da máquina virtual para sua correta execução. A ideia básica da para-virtualização é tornar o sistema operacional que executa na máquina virtual ciente do ambiente virtualizado, podendo assim cooperar com a plataforma de virtualização para obter um melhor desempenho. Um exemplo disso é a exposição de múltiplos relógios de sistema, um real e um virtual, permitindo que o sistema operacional da máquina virtual possa prover estimativas de tempo mais precisas para tarefas que dependem desse recurso, como estimativas de tempo para detectar temporizações críticas do *kernel*.

Na arquitetura x86, o Xen utiliza os 4 níveis de privilégios oferecidos pelo processador (também chamados de *rings*) para ter o controle total sobre a máquina física e limitar as atividades que podem ser realizadas por cada máquina virtual. O hypervisor é executado no nível com maior privilégios, que é chamado de *ring #0*. O sistema operacional e as aplicações de cada máquina virtual são executados nos *rings #1 e #3* respectivamente, sendo o *ring #3* o nível com menos privilégios. O processador é responsável por limitar a execução de determinadas instruções de máquina em alguns níveis de privilégios específicos. Utilizando essa funcionalidade, o Xen é capaz de capturar todas as instruções privilegiadas executadas pelas máquinas virtuais e tratá-las de modo a manter a segurança e o isolamento entre todas as máquinas virtuais.

[2] A para-virtualização é um título recente para uma técnica originalmente implementada pelo VM/370 da IBM, na década de 70 [26].

O controle do tempo que cada máquina virtual utiliza a CPU é feito pelo escalonador de CPUs virtuais (vCPUs) do Xen, que fornece abstrações para relacionar vCPUs às CPUs reais (em processadores com mais de um núcleo) e também para definir a porção de tempo a ser utilizada por cada vCPU. Isso permite isolar o desempenho em termos de CPU entre cada máquina virtual, atribuindo limites máximos para o tempo que cada uma delas pode utilizar a(s) CPU(s) do sistema.

A gerência de memória em sistemas que oferecem virtualização completa, como por exemplo no Denali [66], normalmente é controlada completamente pelo VMM. Tratar todos os acessos à memória é um custo a mais em termos de processamento e por esse motivo os desenvolvedores do Xen optaram por um modelo de gerência de memória simplificado. No Xen, muitas das atividades relacionadas ao acesso à memória são realizadas pelo próprio sistema operacional da VM. No entanto algumas operações são validadas pelo VMM, como por exemplo atualizações na tabela de páginas, para garantir a segurança e o isolamento entre VMs.

Modelo de dispositivos de E/S do Xen

Assim como para CPU e memória, o Xen também possui uma interface diferenciada para o controle do acesso aos dispositivos de entrada e saída, como dispositivos de bloco e/ou NICs. Ao invés de oferecer versões virtualizadas dos drivers de dispositivos às máquinas virtuais, como é feito por alguns VMMs, a abordagem do Xen é prover uma interface genérica de acesso aos dispositivos para cada VM. Nesse modelo de E/S o Dom0, que tem acesso direto a todo o hardware do sistema virtualizado, utiliza o driver real para acessar os dispositivos de hardware e fornece uma interface de comunicação entre os dispositivos e as demais máquinas virtuais. Isso é feito com a exposição de dispositivos genéricos às VMs como, por exemplo, uma NIC genérica ou um dispositivo de armazenamento genérico. A vantagem desse modelo é que apenas o sistema executado no Dom0 precisa ter suporte aos dispositivos físicos para que os recursos dos mesmos sejam disponibilizados para todas as outras máquinas virtuais.

Os módulos que implementam a conexão entre o dispositivo real e o dispositivo visto pela máquina virtual no Xen são chamados de *backend driver* e *frontend driver*. O primeiro é executado pelo Dom0, enquanto o último é executado por cada máquina virtual, conforme mostrado na figura 2.3. A comunicação entre esses dois módulos é feita de forma assíncrona utilizando uma porção de memória que é compartilhada entre o Xen e as máquinas virtuais, chamada de *device channel*. Quando uma VM deseja acessar um dispositivo de E/S, ela utiliza o *frontend driver* para se comunicar diretamente com o *backend driver*, que intermedeia o acesso das VMs ao dispositivo de

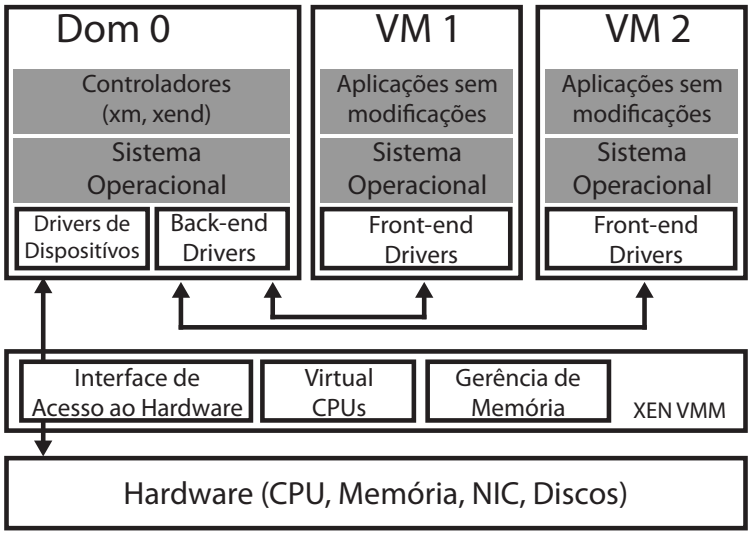


Figura 2.3. Modelo de dispositivos de E/S no Xen

hardware utilizando o driver real instalado no Dom0. Após a requisição ser atendida pelo Dom0, o caminho inverso é utilizado para a resposta à requisição. Esse modelo de E/S é conhecido como *Split Driver Model*.

Modelo de Rede do Xen

Considerando o foco deste trabalho no isolamento de tráfego de rede, é de especial importância observarmos como o Xen implementa o acessa aos dispositivos de rede (*network interface cards, NICs*).

O controle do acesso às NICs presentes no hardware físico é feito de forma semelhante ao que foi descrito na seção anterior, utilizando dois drivers virtuais chamados de *netback* e *netfront*. No entanto, existem diferenças fundamentais entre um dispositivo de rede e um dispositivo de bloco comum, sendo a possibilidade de uma interface de rede receber pacotes que não foram solicitados uma das diferenças fundamentais. Em um dispositivo de bloco, todas as operações realizadas são resultados de requisições feitas por um sistema operacional, não existindo solicitações originadas no próprio dispositivo [14].

Para lidar com essas diferenças, o Xen utiliza interfaces virtualizadas (*vifs*) que podem ser conectadas a um elemento de chaveamento, um *switch* ou roteador virtual, como mostrado na figura 2.4. Cada interface de rede virtual é associada a uma interface de rede de uma máquina virtual e os pacotes são transmitidos utilizando o *device channel*. Essas interfaces são conceitualmente semelhantes à NICs comuns, com espaços de armazenamento temporário para a recepção e transmissão de pacotes de rede. O

elemento de chaveamento deve ser capaz de associar pacotes recebidos com cada VM, a fim de estabelecer a conectividade de cada máquina. Assim como as interfaces virtuais, as interfaces físicas (*physical interfaces — pethX*) também podem estar ligadas ao *switch* virtual. Cada interface física está associada a uma NIC específica e pode ser compartilhada por todas as *vifs*. A escolha de quais interfaces físicas serão utilizadas pelas máquinas virtuais e qual o tipo de tráfego (como por exemplo tráfego de acesso a unidades de disco remotas ou tráfego Web) que deve ser transmitido por cada uma é de responsabilidade do administrador da rede. Essas regras são especificadas no Dom0, que é o único ambiente que tem acesso a essas interfaces. Além disso, o Dom0 também é responsável por receber e repassar os pacotes advindos da rede física utilizando o driver real de cada NIC.

O Xen suporta dois modos de operação para a conectividade entre as máquinas virtuais e as interfaces de rede reais, o modo ponte (*bridge*) e o modo roteado. Ambos os modos de operação utilizam os mecanismos do sistema operacional do Dom0 para encaminhar os pacotes, sendo a principal diferença o nível de exposição das máquinas virtuais à rede física. No modo ponte, todas as máquinas virtuais pertencem à mesma rede da qual a máquina física também faz parte, enquanto que no modo roteado elas podem pertencer redes distintas.

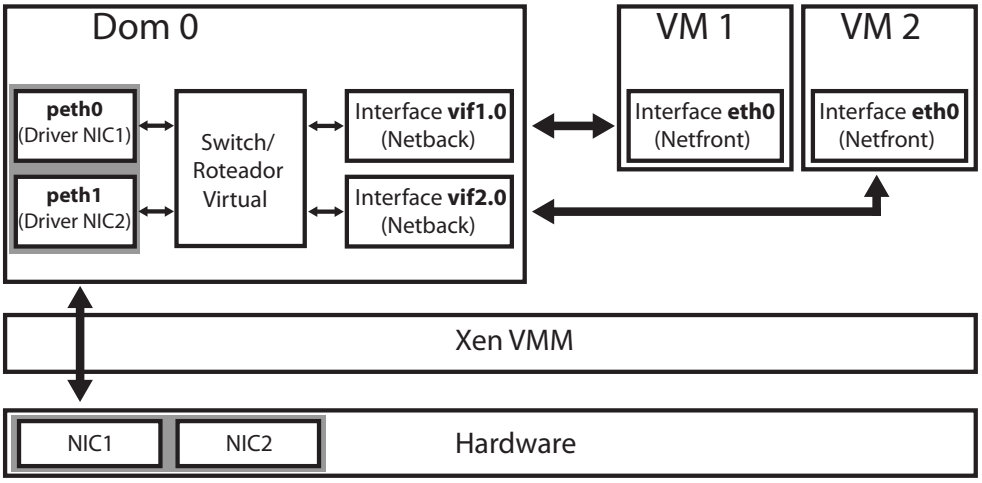


Figura 2.4. Modelo de rede no Xen

O funcionamento básico da transmissão e recepção de pacotes é o mesmo nos dois modos de operação. Quando uma máquina virtual deseja transmitir um pacote, esse pacote é enviado pelo sistema operacional da VM por sua interface de rede (*netfront*), que possui uma relação direta com uma interface virtual residente no Dom0 (*netback*). Assim que o pacote é transmitido pela interface da VM, ele passa pelo *device channel*

para então ser recebido pela interface virtual correspondente. Quando esse pacote chega ao switch/roteador do Dom0, ele é verificado e encaminhado para a interface de destino, que pode ser tanto uma interface real (pethX) ou uma outra interface virtual, no caso de uma comunicação entre VMs na mesma máquina física. Na recepção de pacotes o caminho inverso é percorrido, sendo o *switch* ou roteador virtual o responsável por decidir para qual máquina virtual o pacote recebido deve ser encaminhado.

2.2 Controle de Tráfego

Enquanto soluções de virtualização já garantem uma divisão adequada no acesso ao hardware, especialmente em termos de CPU e memória, utilizando algoritmos de escalonamento de VMs, o mesmo não acontece para o acesso aos recursos de rede. Esta seção discute o problema de controle de tráfego de rede com fins a garantir uma utilização justa dos recursos da rede do ponto de vista dos diversos fluxos de comunicação existentes. Os recursos de rede podem ser representados como frações da banda disponível em cada enlace da rede, ou como espaço nas filas associadas às interfaces dos elementos de chaveamento, como roteadores e *switches*.

O modelo de serviço de rede predominante em redes de comutação de pacotes é o de melhor esforço (*best effort*), que trata cada pacote transmitido com a mesma prioridade e não oferece qualquer abstração de circuitos de rede ou fluxos com garantias de tráfego [45].

No caso de *switches* e roteadores, o controle da alocação de recursos de rede para diferentes tipos de tráfego é feito pela alocação do espaço nas filas associadas a cada interface de rede, utilizadas para armazenar os pacotes que esperam para serem transmitidos. Esse armazenamento temporário é utilizado para evitar o descarte de pacotes caso a rede apresente contenções ou gargalos, como mostrado na figura 2.5. No cenário apresentado, se os transmissores estiverem enviando tráfego à velocidade máxima permitida por seus enlaces de acesso, o roteador representado na figura não será capaz de repassar os pacotes na velocidade em que eles chegam. Os pacotes devem então ser armazenados temporariamente para serem enviados à velocidade permitida pelo enlace de saída do elemento de chaveamento. Quando os recursos de um determinado elemento da rede se esgotam, ocorrem congestionamentos.

O congestionamento em um enlace da rede afeta todas as aplicações que dependem desse enlace para trocar dados. Uma das formas de resolver o problema seria distribuir os recursos disponíveis para as aplicações até o limite da capacidade do enlace de rede, bloqueando o tráfego das demais aplicações até que uma parcela dos recursos

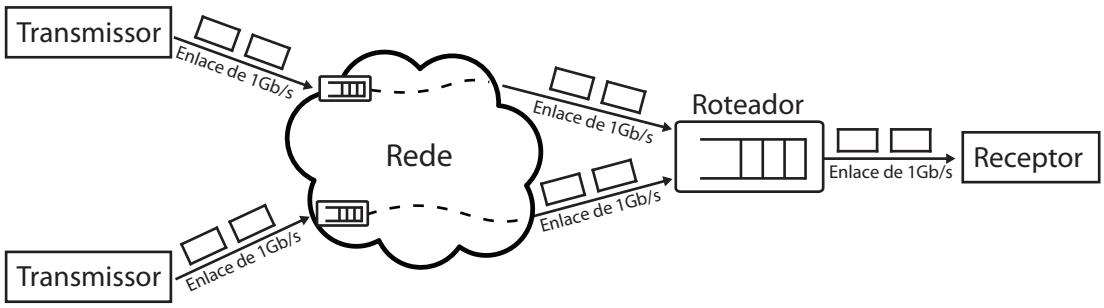


Figura 2.5. Um gargalo na rede que pode levar a um congestionamento

seja liberado. Porém, em uma rede onde todas as aplicações tem igual prioridade de uso, é preferível uma solução capaz de garantir uma divisão justa dos recursos entre todas as aplicações.

Existem diversas soluções para o controlar a alocação de recursos de rede de forma a evitar e resolver congestionamentos e essas soluções podem ser classificadas de diversas maneiras. Segundo [45], três dos fatores que diferenciam essas soluções são:

- **Ponto de implementação da solução:** As soluções para controle de tráfego podem ser destinadas a roteadores e *switches* presentes no núcleo da rede, ou aos servidores nas bordas da mesma. Nesse último caso as soluções são normalmente implementadas na camada de transporte da pilha de protocolos de rede. O uso dessas soluções não é mutuamente exclusiva, podendo haver uma cooperação entre elas para realizar o controle da alocação dos recursos disponíveis de forma mais eficiente.

- **Forma de alocação de recursos:** Existem basicamente duas formas alocação. A primeira delas é baseada em reserva de recursos, onde cada fluxo na rede deve primeiro reservar uma certa quantidade de recursos que serão utilizados por ele, para depois iniciar a troca de dados. O outro modo é baseado em notificações, onde os fluxos utilizam os recursos sem reserva prévia e ajustam o seu consumo de acordo com o estado da rede, que é monitorado continuamente.

- **Mecanismo de controle:** Soluções de alocação de tráfego podem ser baseadas em janelas deslizantes ou medições de banda. Algoritmos de janela deslizante são utilizados para controlar a quantidade de pacotes que são colocados na rede a cada instante. Esse mecanismo é mais comum em mecanismos centrados em servidores, pois permite avaliar o consumo de recursos da rede em termos da quantidade de pacotes em função do tamanho da janela utilizada. Soluções baseadas em medição de banda determinam a alocação de recursos a cada fluxo diretamente

como um valor de banda permitida, que deve ser monitorada e controlada por alguma forma de implementação adequada.

Na implementação de soluções para alocação de recurso em rede, certas combinações desses fatores são mais adequadas que outras, dadas as características envolvidas em cada caso.

Soluções baseada em janela deslizante são normalmente mais adequadas para implementação nas bordas da rede, já que oferecem uma forma simplificada de representar a alocação, especialmente para o ajuste baseado em notificações. O controle de congestionamento utilizado pelo protocolo TCP é considerado um exemplo canônico desse tipo de solução. Já soluções baseadas em medições de banda são mais adequadas para implementação no núcleo da rede, onde medições mais precisas, utilizando relógios e contadores em hardware, são normalmente possíveis. Essas soluções se prestam bem à operação com reserva prévia de recursos, apesar dessa reserva poder variar ao longo do tempo. Esse tipo de solução é normalmente caracterizada por políticas de escalonamento utilizadas nas filas dos elementos de chaveamento da rede. Ambas as soluções são discutidas a seguir, pois alguns elementos de ambas são importantes para a solução adotada por este trabalho.

2.2.1 Controle de congestionamento do TCP

Em seu início, com o crescimento do número de computadores e da demanda pelo uso da rede, a Internet começou a sofrer um "colapso de congestionamento". Naquela época, sem qualquer controle de congestionamento, o protocolo TCP fazia uso da rede como se o problema não existisse, retransmitindo pacotes que eram perdidos à taxa que cada máquina conseguia enviar e agravando ainda mais a situação da rede. Para resolver esse problema, o protocolo foi alterado com a adição de mecanismos de controle de congestionamento [29].

O princípio do controle de congestionamento de TCP é fazer com que cada máquina nas extremidades de uma conexão (*end host*) avalie constantemente quantidade de recursos disponíveis na rede de forma a determinar qual o volume de dados que que pode ser introduzido na rede. Em TCP, essa determinação é usada para se definir uma janela de transmissão, um certo volume de dados que pode estar em trânsito na conexão em qualquer instante. Ao enviar dados que preencham completamente essa janela, a conexão TCP considera que já utilizou a quantidade de recursos a que tem direito. Nesse momento, um *end host* deve então esperar que os recursos consumidos voltem a ficar livres para que ele possa enviar mais dados. Isso é feito através do recebimento de mensagens de confirmação de recebimento (*acknowledgements*, ACKs) enviados pelo

destinatário. Essas confirmações indicam para o transmissor que quais pacotes que faziam parte da janela de transmissão foram recebidos pelo destinatário — consequentemente, não estando mais em trânsito. Nesse momento, o transmissor pode atualizar a sua visão da janela de transmissão, retirando dela os dados confirmados, o que abre espaço para que mais dados sejam transmitidos. À medida que mais confirmações são recebidas, mais dados podem ser enviados, até que a janela seja completamente preenchida. A janela dessa forma "desliza" sobre os dados a serem transmitidos, daí seu nome. Por esse padrão de controle, onde o fluxo de pacotes é dependente dos ACKs enviados pelo receptor, diz-se que TCP é auto-temporizado *self-clocked*.

Apesar de o funcionamento do TCP ser simples, a tarefa de determinar a quantidade de recursos de rede disponíveis não é uma tarefa fácil, pois a rede é um recurso distribuído e compartilhado entre vários servidores. A quantidade de recursos disponíveis é variável e depende de diversos fatores, como as rotas tomadas por cada fluxo e a quantidade de tráfego gerado pelos demais servidores. TCP utiliza dois elementos principais para avaliar essa disponibilidade: um mecanismo para avaliar os recursos disponíveis no início de uma conexão, denominado *slow start* (partida lenta), e um mecanismo para ajustar a taxa de transmissão ao longo do tempo, depois que a conexão se estabiliza, denominado *congestion avoidance*[3] Um princípio importante adotado por TCP é que perdas de pacotes devidas a erros são relativamente raras, dada a qualidade dos meios de transmissão atuais. Sendo assim, a causa preponderante para a perda de pacotes é o descarte devido a congestionamento. TCP, então, interpreta retransmissões de pacotes perdidos com um indicador de congestionamento.

2.2.1.1 *Slow Start*

No momento do estabelecimento de uma conexão, os *end hosts* não têm uma forma simples de determinar a banda disponível para a mesma. O mecanismo de partida lenta foi desenvolvido como uma forma de se garantir que as conexões não fossem iniciadas com taxas de transmissão arbitrárias que pudessem causar grande contenção na rede. Conexões devem, ao contrário, inciar com uma taxa de transmissão baixa (uma janela mínima) e ir aumentando essa taxa (aumentando a janela) à medida que verificam que não há congestionamento.

Na fase de *slow start*, o mecanismo de *self clocking* tem um comportamento especial. A cada vez que todos os dados em uma janela de transmissão são confirmados,

[3]O dicionário Houaiss já reconhece o neologismo evitação/evitamento, que seria a tradução mais adequada para a palavra *avoidance* [3].

a janela de transmissão dobra de tamanho[4] e o processo se repete até que perdas sejam detectadas. Como discutido anteriormente, TCP interpreta tais perdas como um indicador de que o ponto de congestionamento tenha sido atingido. Nesse momento, por precaução, a janela é reduzida pela metade e esse valor é escolhido com o valor ideal da janela naquele momento (determinando uma certa taxa de transmissão em função do tempo que os ACKs levam para chegar).

2.2.1.2 *Congestion Avoidance*

Uma vez determinada uma primeira taxa ideal ao final do *slow start*, o emissor tenta continuamente utilizar mais recursos da rede a cada confirmação (ACK) dos receptores, a fim de identificar momentos em que recurso de rede podem ter sido liberados por outros fluxos. Ao contrário da fase inicial, entretanto, a janela de transmissão (que determina a taxa) é incrementada mais lentamente, um pacote a cada janela completa transmitida com sucesso. Esse crescimento, apesar de conservador, em algum momento levará a uma nova situação de congestionamento. Nesse caso, entretanto, partindo da premissa de que uma janela só é incrementada em um pacote quando toda a janela foi transmitida com sucesso, a expectativa é que apenas esse novo pacote seja perdido por congestionamento. TCP então reduz a janela novamente pela metade e retorna ao comportamento de tentar incrementar periodicamente a janela. Esse mecanismo é conhecido como *incremento aditivo-decremento multiplicativo* (*additive increase-multiplicative decrease, AIMD*). Esse tipo de comportamento é reconhecido, na teoria de controle, como adequado para manter a estabilidade do sistema [29].

Essas características tornam o controle de congestionamento do TCP altamente escalável, pois ele não depende de nenhum auxílio dos roteadores para controlar o seu uso da rede. Em particular, cada conexão controla sua taxa independentemente, sem depender de conhecimento sobre as demais conexões na rede.

2.2.2 Alocação de Recursos da Rede

Com o mecanismo de *congestion avoidance*, o TCP garante uma divisão justa entre todos os fluxos com demandas por recursos da rede. Porém, ao administrar uma rede compartilhada por diversos serviços com diferentes características de tráfego de diferentes necessidades, também é importante que o administrador possa definir prioridades para diferentes tipos de tráfego.

[4]Na prática, a janela não dobra em um só instante, mas esse crescimento é distribuído a cada ACK recebido. Esse comportamento, de dobrar a janela a cada vez, leva a um crescimento exponencial, que está longe de ser considerado "lento". Entretanto, por permitir um crescimento gradativo, ao invés de iniciar com um valor arbitrariamente alto, considera-se o nome *slow start* apropriado nesse caso.

Alguns dos exemplos mais comuns de aplicações que apresentam necessidades específicas são aplicações interativas, como telnet ou navegadores Web, e aplicações de tempo real. A alocação diferenciada de recursos de rede também pode ser vista do ponto de vista comercial. Em uma infraestrutura de um provedor de recursos computacionais por exemplo, um dos clientes pode estar disposto a realizar um investimento maior para ter um serviço diferenciado. Como um outro exemplo, um provedor de Internet residencial pode querer oferecer diversos planos com diferentes garantias de banda para seus clientes.

Esse tipo de alocação diferenciada exige uma visão mais abrangente da situação da rede e por isso devem considerar sua implementação dentro da rede, nos elementos de chaveamento. Nesse caso é possível considerar soluções baseadas na reserva de recursos e no estabelecimento explícito de garantias de taxas de transmissão. Como mencionado anteriormente, soluções implementadas dentro da rede normalmente são definidas por algoritmos de controle das filas dos roteadores e *switches* da rede que permitam maior controle sobre a alocação de banda.

Esses algoritmos, também chamados de disciplinas de fila (*queuing discipline*) são compostos por uma disciplina de escalonamento (*scheduling discipline*) e uma política de descartes (*drop policy*). Essas tem a função de controlar tanto a largura de banda de cada fluxo quanto o seu uso do espaço para armazenamento temporário, respectivamente. O controle da largura de banda é feita através das decisões de quais pacotes que devem esperar na fila ou serem transmitidos e as regras para o uso do armazenamento temporário definem quais pacotes devem ser descartados ao invés de serem armazenados.

A disciplina de fila mais simples é a FIFO (*first-in, first-out*). As regras envolvidas nessa disciplina definem que novos pacotes sempre entram no fim da fila e que o próximo pacote a ser enviado é sempre aquele no início da fila. Essa disciplina também é conhecida como "FIFO com descartes da calda" (*FIFO with tail drop*), onde FIFO é a sua disciplina de escalonamento e descartes da calda a política de descartes. Por ser muito simples e necessitar de poucos recursos de hardware, quase todos os roteadores da Internet possuem uma implementação dessa disciplina [45]. No entanto, essa disciplina não oferece qualquer diferenciação de serviço entre fluxos de apresentam diferentes prioridades.

Qualquer disciplina capaz de tratar de forma diferenciada os diferentes fluxos depende da capacidade de identificação desses fluxos. Isto é, cada pacote recebido pelo elemento de chaveamento deve poder ser associado ao fluxo a que pertence. Isso é normalmente feito pela identificação de um padrão nos cabeçalhos do pacote, como endereços de origem e destino. Uma vez que um pacote seja associado ao seu fluxo,

o algoritmo de processamento da fila pode decidir como tratar esse pacote em relação aos demais em função do estado anterior associado ao fluxo que foi identificado.

Uma variação da politica FIFO para que ela seja capaz de diferenciar fluxos de diferentes prioridades é chamada fila de prioridades (*priority queuing*). Essa variação consiste em utilizar várias filas com disciplina FIFO de forma que fluxos de tráfego com diferentes prioridades são encaminhados para diferentes filas. A disciplina de escalonamento define que enquanto uma fila de maior prioridade não estiver vazia fila ela é sempre atendida antes das outras de menor prioridade. Essa ideia pode ser considerada um simples passo além da disciplina FIFO e não pode ser considerada uma boa solução para diferenciação de tráfego. O problema com a fila de prioridades é a possibilidade de *starvation* das filas de menor prioridade caso um determinado fluxo de tráfego esteja sempre presente no roteador.

Um problema que afeta tanto a disciplina de fila de prioridades quanto a FIFO é que pacotes que são alocados para a mesma fila são tratados igualmente, sem qualquer distinção entre pacotes de fluxos diferentes. Um fluxo com maior volume de tráfego pode consumir uma maior parcela dos recursos de rede, restringindo a divisão justa dos recursos disponíveis. Uma disciplina de fila que tenta resolver esse problema é a disciplina *Fair Queuing* (FQ).

A ideia de FQ é dividir diferentes fluxos de mesma prioridade em diversas filas, utilizando para isso os endereços fonte e/ou destino de cada um. Os pacotes dessas filas seriam então transmitidos seguindo um algoritmo *round-robin*. Quando um fluxo com grande volume de tráfego utiliza todos os recursos disponíveis da sua respectiva fila, seus pacotes começam a ser descartados. Assim como nas outras disciplinas de fila, o descarte de pacotes é utilizado como forma de informar indiretamente o servidor que os recursos da rede se esgotaram e que o controle de congestionamento implementado nos servidores deve ser ativado.

Uma importante variação da disciplina FQ é a Weighted Fair Queuing (WFQ). Essa disciplina herda quase todas as características da FQ, porém ela oferece a possibilidade de atribuir pesos a cada uma das filas que serão atendidas com o algoritmo de *round-robin*. Utilizando essa disciplina é possível, por exemplo, dividir os recursos do roteador de forma que um dos usuários utilize duas vezes mais recursos que os outros.

Uma propriedade comum às duas disciplinas, FQ e WFQ, é que elas são capazes de realizar um escalonamento com conservação de trabalho (*work-conserving*). Isso significa que caso uma das filas não esteja utilizando os recursos de rede da interface, a disciplina pode repassar esses recursos automaticamente para alguma outra fila que apresente demandas por recursos. Dessa forma o escalonamento com conservação de trabalho pode melhorar a utilização média da rede mantendo a divisão de acordo com

as regras definidas pelo operador da rede quando todas as filas apresentam demanda por recursos.

Até aqui apresentamos disciplinas de fila que podem ser utilizadas para priorizar tráfego e dividir os recursos de cada roteador de forma justa. Uma outra forma de realizar o controle de tráfego é através da especificação de uma largura fixa de banda para cada fluxo. Um dos algoritmos que podem ser utilizados para controlar o tráfego especificando uma velocidade de transmissão é o *token bucket*. A ideia do algoritmo é apresentada na figura 2.6. Cada fluxo é associado a uma fila e cada fila deve manter um balde que é preenchido com *tokens* à uma taxa constante. A medida que novos pacotes chegam à fila para serem transmitidos, a fila verifica se existem *tokens* disponíveis no balde. Caso a resposta seja positiva, o pacotes é encaminhado e a quantidade de *tokens* referente ao tamanho do pacote transmitido é removida do balde. Caso não existam *tokens* disponíveis no balde de *tokens*, o pacote é mantido na fila caso exista espaço disponível para seu armazenamento ou é descartado, caso a fila já esteja cheia.

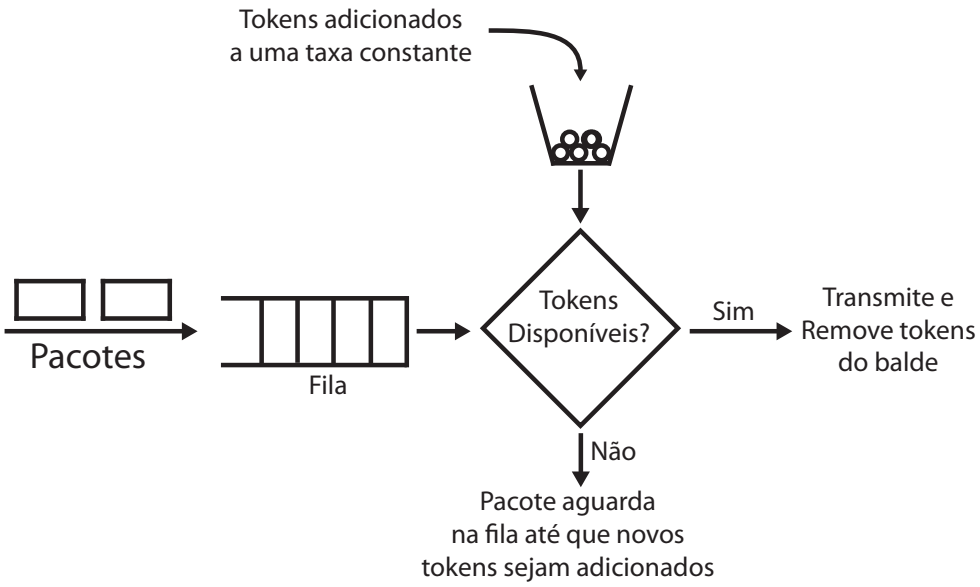


Figura 2.6. Funcionamento do algoritmo *Token Bucket*

Uma característica importante do algoritmo *token bucket* é relacionada à sua capacidade de armazenar buckets no balde enquanto não há tráfego para ser enviado. Isso permite que pacotes pertencentes a fluxos com padrões de tráfego de rajadas (*bursts*) de alta velocidade, como é o tráfego da Internet, não sejam descartados imediatamente. O tamanho do balde é uma forma de ajustar qual é a duração da rajada permitida pelo algoritmo *token bucket*. A robustez do algoritmo tornou-o o algoritmo mais utilizado para controle de tráfego baseado em larguras de banda.

2.2.3 Controle de Tráfego no Linux

O sistema operacional Linux possui implementações de muitas das disciplinas de fila discutidas na seção anterior. Isso faz com que o Linux possa ser utilizado como um sistema para roteadores de pequeno porte que oferecem diferenciação de serviço para o tráfego de rede. Antes de iniciar a discussão sobre as disciplinas de fila do Linux, é necessário conhecer o fluxo percorrido pelos pacotes dentro desse sistema. A figura 2.7 apresenta uma visão geral desse fluxo, desde a recepção até a transmissão. Quando um pacote chega por uma interface de rede, o Kernel verifica o destino desse pacote e decide por passá-lo para as camadas superiores da pilha de protocolos de rede ou então encaminhá-lo para o redirecionamento e, nesse último caso, o pacote segue para uma das interfaces de saída. Em ambos os sentidos, na entrada e na saída de pacotes, é possível adicionar as disciplinas de fila para oferecer um tratamento diferenciado do tráfego de rede.

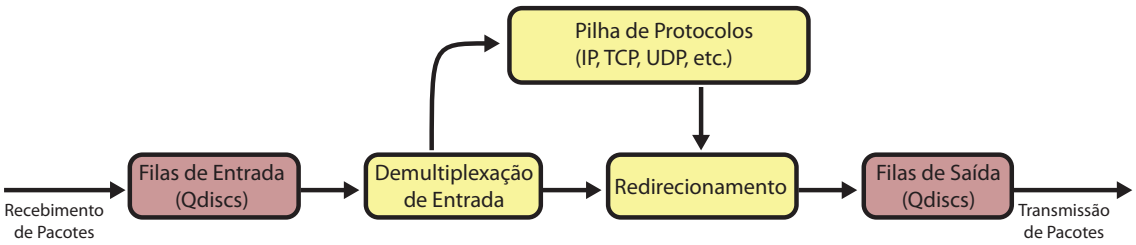


Figura 2.7. Tratamento de rede do Kernel Linux desde a recepção de dados até o envio.

O elemento principal na implementação de disciplinas de fila no Linux é chamado de *Queuing Discipline*, ou (*qdiscs*). As várias opções de politicas de fila no Linux são divididas em duas categorias:

- Disciplinas *classful*: Disciplinas que podem ser combinadas utilizando o conceito de *classes*. Uma classe é uma implementação de uma disciplina de fila que é capaz de repassar pacotes à outras classes. O Fluxo de pacotes entre cada classe é definido através de filtros de classificação.

- Disciplinas *classless*: Essas disciplinas tratam todos os pacotes recebidos de forma semelhante. São implementações mais simples que normalmente são utilizadas em conjunto com disciplinas *classful*. No Linux, a disciplina de fila FIFO por exemplo é implementada como uma disciplina *classless*.

Essas duas categorias de disciplinas podem ser combinadas de forma a criar uma hierarquia de disciplinas de fila, conforme apresentado na figura 2.8. Os filtros utiliza-

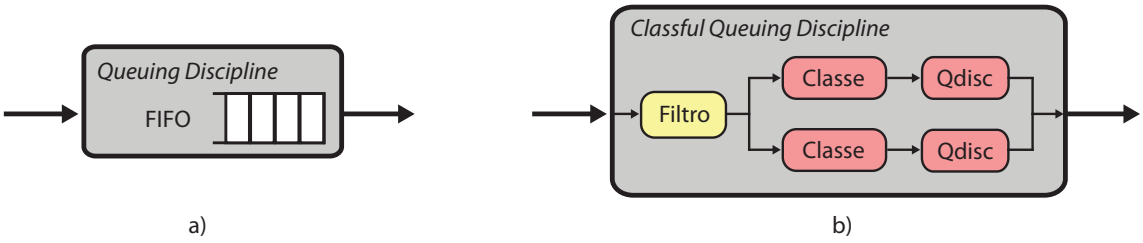


Figura 2.8. Implementação das Disciplinas de Fila no Linux.

dos para classificar diferentes tipos de tráfego de rede são especificados em termos dos valores dos campos dos pacotes, como endereço IP ou Porto de origem/destino.

Cliente	Tipo do tráfego	Limite Individual	Limite Cliente
A	Tráfego Web	400Kbps	700Kbps
A	Tráfego P2P	300Kbps	
B	Livre	Livre	300Kb

Tabela 2.1. Exemplo de uso do HTB: Garantias a serem definidas para 3 classes de tráfego

Dentre as disciplinas de fila disponibilizadas pelo Kernel Linux, a disciplina *classful Hierarchical Token Bucket* [16] (HTB) é uma das importantes para o entendimento desse trabalho pois foi utilizada na implementação pela solução estudada. O HTB é uma implementação do algoritmo *token bucket* de forma hierárquica, oferecendo uma vasta gama de possibilidades para o controle de tráfego de rede. Para entender melhor como funciona essa disciplina de fila, considere o seguinte exemplo. Suponha que um operador de rede gerencie um enlace de 1 Mbps e precisa oferecer diferentes garantias de envio de dados para dois clientes A e B. Além disso, os clientes precisam de garantias específicas para diferentes tipos de tráfego, conforme a tabela 2.1.

Toda configuração do HTB é feita utilizando *classes* que são organizadas em forma de uma árvore. Cada uma das *classes* pode ser configurada utilizando duas variáveis, `rate` e `ceil`, que controlam a vazão do tráfego de cada classe. A variável `rate` é tratada pelo HTB como um *limite mínimo* para a vazão de cada classe e a variável `ceil` é o *limite máximo*. Isto é, a taxa de envio de cada classe é limitada pela variável `rate`, porém, caso a demanda por recursos de rede seja maior que `rate`, a classe pode "pegar tokens emprestados" das suas classes pai até que sua vazão atinja o limite de `ceil`. A configuração desses dois limites é útil para oferecer um escalonamento com conservação de trabalho com um limite superior, através do que é chamado empréstimo de tokens entre classes. Uma forma de configurar as classes da disciplina de fila HTB para oferecer as garantias especificadas na tabela 1 é mostrada na figura 2.9.

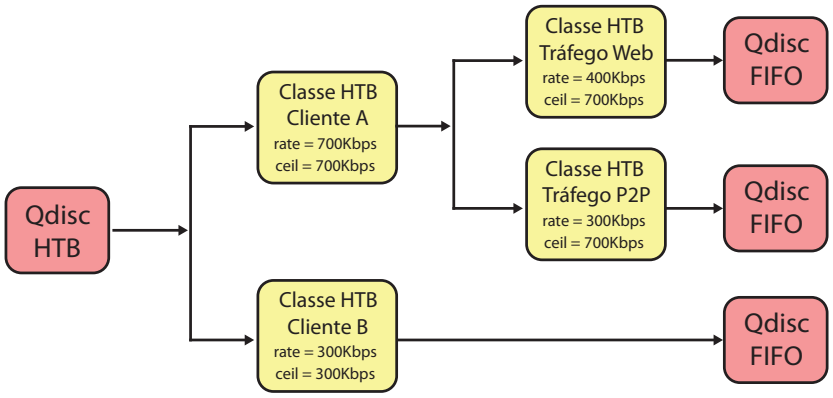


Figura 2.9. Exemplo de configuração da disciplina de fila HTB para atender as garantias de tráfego especificadas na tabela 2.1.

2.3 Modelo de Rede de um Datacenter

As tradicionais arquiteturas de rede para datacenters são baseadas em um modelo conhecido como arquitetura de três camadas (*three-tier architecture*) [12]. A camada inferior, conhecida como camada de acesso (*access tier/edge tier*), interliga os servidores do datacenter utilizando *switches* de acesso através dos enlaces de acesso. Cada um desses *switches* são conectados a outros pertencentes à camada superior, denominada camada de agregação (*aggregation tier*). Por fim, os *switches* localizados na camada de agregação são interconectados através dos vários *switches* presentes na camada núcleo (*core tier*). Cada uma das ligações entre uma camada inferior e a camada imediatamente superior podem contar com mais de um enlace de rede por redundância, garantindo maior disponibilidade da rede.

Nessas redes também existem roteadores, responsáveis por fazer a ligação dos servidores do datacenter com a Internet, normalmente conectando-se aos *switches* da camada núcleo. Entretanto, a maior parte dos elementos da rede são *switches* de camada 2 (*Layer 2 switches*), que encaminham pacotes utilizando o protocolo *Ethernet*. O motivo para a predominância desses equipamentos é a sua grande capacidade de processamento com baixo custo, o que reduz o custo total de interligar todos os servidores do datacenter.

O uso de *switches* de camada 2 implica que todos os servidores estão no mesmo domínio de *broadcast* da rede Ethernet. Isso significa que uma inundação (*flood*) na rede, como um *ARP Request* [45] por exemplo, envolverá todos os servidores. Para evitar esse problema e fazer melhor uso dos recursos os servidores são organizados em grupos isolados utilizando-se VLANs [38]. Como o protocolo de camada 2 normalmente utilizado é o Ethernet, um algoritmo de árvore geradora mínima (*minimum spanning*

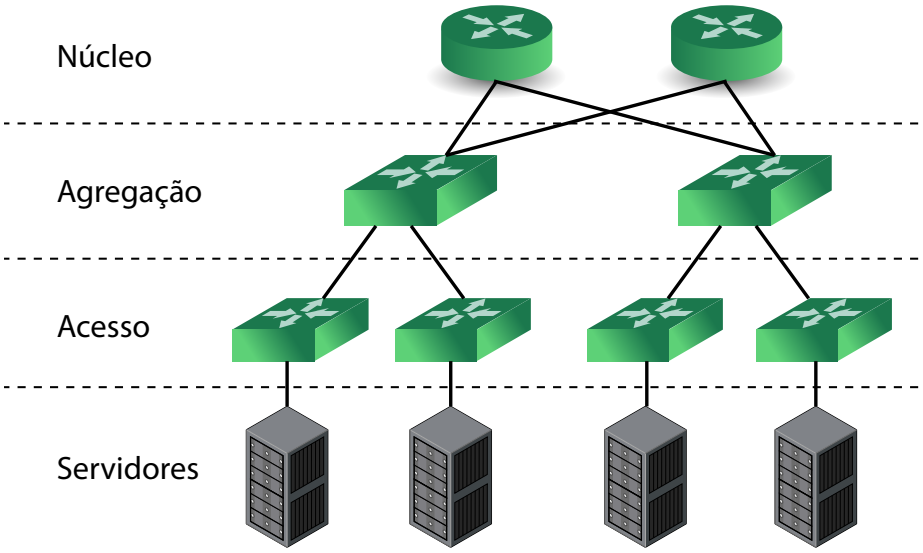


Figura 2.10. Tradicional Arquitetura de Rede de Datacenters em Três Camadas

tree) também é utilizado para remover caminhos redundantes na rede, fazendo com que apenas alguns canais sejam utilizados para todo o tráfego. Por esse motivo, embora a topologia física de um datacenter siga uma arquitetura de três camadas, podendo ser vista como uma floresta (*grafo*) com várias raízes (os *core switches*), na prática os pacotes são roteados de acordo com uma topologia lógica definida por VLANs e árvores geradoras.

A escalabilidade em uma rede com essa arquitetura é obtida através da instalação de *switches* de diferentes capacidades em cada uma de suas camadas. Quanto mais próximo à raiz, maior deve ser a capacidade do switch, que é responsável por processar todo o tráfego trocado entre os *switches* das camadas inferiores. Para aumentar a capacidade da rede, é necessário utilizar *switches* com enlaces de maior capacidade, ou agrupar diversos enlaces para fazer a ligação entre dois *switches*. Essa abordagem é conhecida como escalabilidade vertical (*scale-up*). Um exemplo seria ligar os servidores aos *switches* de acesso com enlaces de 100 Mbps, ligar aqueles *switches* por enlaces de 1 Gbps aos *switches* de agregação, que seriam ligados aos *switches* do núcleo por enlaces de 10 Gbps.

Usualmente, entretanto, a capacidade agregada dos enlaces em um determinado nível é menor que capacidade agregada dos enlaces no nível imediatamente inferior. Isso é denominado *oversubscription* em inglês, indicando que a capacidade de um nível não foi dimensionada para atender ao pior caso de tráfego no nível inferior.

Apesar de ser muito utilizada, devido ao problema de *oversubscription*, a arquitetura de rede em três camadas é considerada um dos principais problemas relacionados

ao desempenho da rede de datacenters. Essa topologia limita severamente a largura de banda de bisseção (*bisection bandwidth*) da rede — largura de banda disponível entre dois pontos quaisquer da rede, considerando o pior caso — quando consideramos os enlaces próximos à raiz da árvore. Esses enlaces apresentam alta *oversubscription*, variando de 10:1 a 80:1 a relação entre a capacidade da rede de acesso e da rede núcleo [19]. Além disso, a topologia não possibilita a escalabilidade horizontal (*scale-out*) — que permite a inserção de mais elementos na rede sem sobrecarregar os já existentes. Prover garantias de tráfego em ambientes que utilizam uma topologia de rede de três camadas é uma tarefa difícil, pois é necessário considerar a capacidade e a demanda de cada enlace da rede para realizar alocação de tráfego considerando as garantias exigidas por cada cliente. Para isso, seria necessário conhecer toda a topologia da rede e quais são os enlaces utilizados por cada fluxo, tornando a tarefa complicada para a escala de um datacenter.

Levando em consideração os problemas da arquitetura de três camadas, vários trabalhos de pesquisa produziram novas topologias de rede para datacenter escaláveis [1, 20–22, 41, 42, 50]. Muitas dessas propostas também apresentam soluções capazes de prover escalabilidade para a largura de banda de bisseção utilizando chaveamento de múltiplos caminhos (*multi-path switching*) [20, 41, 50]. Nesse caso, havendo uma multiplicidade de caminhos em um nível da árvore, todos os caminhos podem ser igualmente utilizados para encaminhar pacotes entre dois pontos quaisquer. Isso não é possível em uma rede tradicional, onde diferentes caminhos teriam que ser atribuídos a diferentes VLANs e o tráfego de uma VLAN não poderia ser transferido para enlaces de outras. Outra forma de reduzir o problema da largura de banda de bisseção é garantir que máquinas virtuais que precisem trocar um grande volume de mensagens sejam colocadas em máquinas físicas próximas na rede de acesso, de forma a minimizar o número de enlaces da rede que cada fluxo atravessa [35, 38].

Tecnologias que reduzem o impacto da largura de banda de bisseção escalável permitem que o problema de problema de controlar o tráfego garantindo as exigências de cada cliente enfoque apenas os enlaces de acesso. Se uma solução de controle de tráfego é capaz de garantir as demandas de cada cliente no seu acesso à rede, garantidamente a rede tem recursos para levar o tráfego até o seu destino. Nesse contexto, este trabalho explora esse fato, focando no problema de controle de tráfego nos enlaces de acesso, considerando que outras soluções existem para resolver o problema da capacidade interna da rede.

2.4 Gatekeeper

Apesar de existirem diversas soluções para realizar o controle do tráfego de rede tanto nos servidores quanto no núcleo da rede, nenhuma dessas soluções conseguem controlar o tráfego provendo garantias de banda de forma intuitiva e precisa no ambiente de um datacenter compartilhado. O controle de congestionamento de TCP, por exemplo, é uma solução para evitar o colapso da rede por congestionamento, mas não fornece quaisquer garantias de alocação de uma certa fração dos recursos da rede, apenas que os recursos serão igualmente divididos entre os fluxos existentes. Além disso, a técnica não funciona na presença de um fluxo "egoísta", que não respeita o princípio de redução da taxa de transmissão na ocorrência de descartes de pacotes.

Por outro lado, disciplinas de fila implementadas no núcleo da rede não são eficientes para controlar o consumo dos recursos se não houver cooperação dos servidores. Além disso, o controle no núcleo da rede não oferece a escalabilidade necessária para realizar uma alocação de recursos específica para cada fluxo da rede. Uma outra estratégia seria utilizar algoritmos limitadores de banda, como o *token bucket*, para controlar a quantidade de tráfego que cada cliente insere na rede. No entanto, essa abordagem não é eficiente para o controle da utilização de recursos por cliente, por dois motivos: primeiro, limitadores não permitem que banda não utilizada por um cliente seja aproveitada momentaneamente por outro cliente em outro servidor; segundo, limitadores na transmissão não resolvem o problema de congestionamento causado quando diversos transmissores, individualmente limitados, direcionam seu tráfego para um mesmo destino, excedendo o limite de tráfego esperado para aquela máquina.

Para solucionar esses problemas, foi proposto o Gatekeeper [54, 55], um mecanismo de controle de tráfego centrado em servidores que utiliza limitadores de banda e um protocolo de controle distribuído para prover isolamento do consumo dos recursos de rede entre os clientes de um datacenter. O Gatekeeper é capaz de realizar o controle de tráfego com base nas garantias especificadas por cada cliente, oferecendo a esses um modelo intuitivo para a especificação da quantidade de recursos que eles necessitam. Nesse modelo de garantias de tráfego, o cliente tem a visão de que há um switch lógico que interliga apenas seu conjunto de máquinas virtuais, sendo a capacidade do enlace que liga o switch às VMs especificado pelo cliente, conforme mostrado na figura 2.11.

Esse modelo é similar a um ambiente que não utiliza virtualização, onde as máquinas físicas são conectadas diretamente a um único *switch*. Por esse motivo, o modelo é bastante intuitivo aos clientes que normalmente contratam os recursos de um datacenter. A abstração do *switch* virtual único deixa claro para o usuário que o desempenho do seu conjunto de máquinas não pode ser prejudicado por qualquer padrão de tráfego

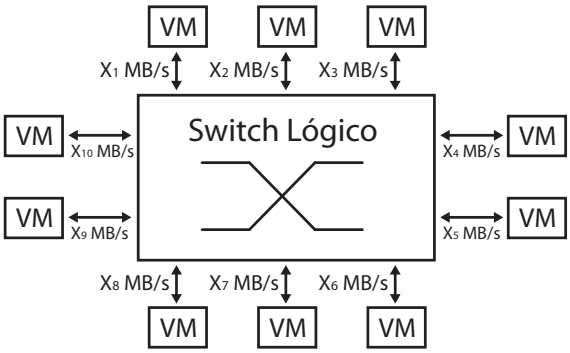


Figura 2.11. Modelo de garantias provido pelo Gatekeeper na visão de um cliente do datacenter.

gerado por outros usuários do datacenter. Entretanto, se o usuário assim desejar, suas VMs podem ainda se beneficiar qualquer banda excedente existente no sistema. Exceto por permitir esse uso de banda excedente acima da taxa garantida em alguns instantes (X_i, na figura 2.11), permitindo um escalonamento com conservação de serviço, esse modelo é semelhante ao *hose model* [17, 20].

Como mencionado anteriormente, o Gatekeeper assume que a largura de bisseção da rede é escalável e concentra-se no gerenciamento dos enlaces de acesso, que ligam as máquinas físicas à rede do datacenter. Cada enlace de acesso pode ser compartilhado por diversos serviços associados a VMs de diferentes clientes. O Gatekeeper provê um controle de tráfego eficiente para essas VMs, evitando que a demanda excessiva por tráfego de um dos clientes, tanto na transmissão quanto recepção de tráfego, prejudique o desempenho de rede dos demais.

Detalhes específicos sobre a implementação original do Gatekeeper podem ser encontrados na dissertação de Paolo Victor Soares [54]. A discussão nesta seção aborda os aspectos significativos para a descrição das mudanças envolvidas no desenvolvimento da nova solução, o Gatekeeper-ng, foco deste trabalho.

2.4.1 Arquitetura

A figura 2.12 apresenta uma visão geral da arquitetura do Gatekeeper. O sistema intercepta os pacotes enviados pelas máquinas virtuais locais destinados à interface de rede do servidor e os pacotes recebidos pela mesma interface de rede destinados às máquinas virtuais local, aplicando mecanismos de controle em ambas as direções para obter o controle adequado do tráfego. Internamente, o sistema possui três componentes: o escalonador de saída, o escalonador de entrada e o agente de congestionamento. Em sua implementação original, esses elementos atuavam sobre o tráfego entre o *switch*

virtual (linux *bridge*, nesse caso) e a interface física da máquina.

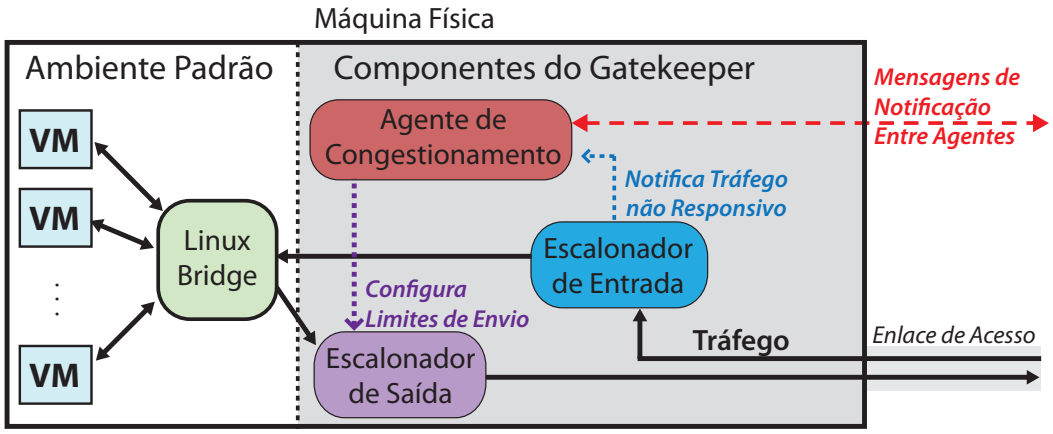


Figura 2.12. Arquitetura do Gatekeeper

O princípio geral de operação do sistema é que o agente de congestionamento determina taxas de transmissão permitidas para cada VM, que são controladas pelo escalonador de saída. O escalonador de entrada monitora o tráfego recebido, detectando situações em que o tráfego recebido viole a banda garantida para aquela máquina no modelo do *switch* único. Nesse caso, o escalonador de entrada descarta pacotes do fluxo excedente e pode enviar mensagens para os escalonadores de saída das máquinas de onde se origina o tráfego em excesso.

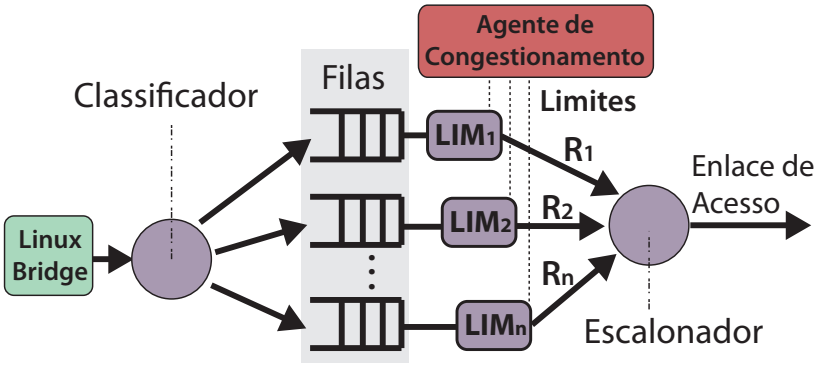


Figura 2.13. Escalonador de Saída do Gatekeeper

O escalonador de saída é detalhado na figura 2.13. Esse componente controla o tráfego de saída utilizando um algoritmo de escalonamento de pacotes com garantias de envio (como o *WFQ*, discutido na seção 2.2.2, porém utilizando limites fixos ao invés de pesos). Os pacotes a serem transmitidos são organizados em múltiplas filas e cada uma dessas filas é associada a uma máquina virtual. O algoritmo de escalonamento envia os pacotes armazenados em cada uma das filas de acordo com a taxa de envio

definida pelo agente de escalonamento para aquele fluxo, R_i. Caso uma das filas esteja vazia, o algoritmo de escalonamento distribui a banda ociosa entre as filas que possuem pacotes a serem enviados.

O escalonador de entrada é apresentado na figura 2.14. Uma tarefa essencial desse módulo é detectar casos em que o fluxo agregado de todos os transmissores direcionado a uma VM extrapole o limite garantido para aquele usuário e prejudique o comportamento de outras aplicações. Como esse tráfego excedente pode causar congestionamento nos enlaces da rede, antes de atingir a máquina física de destino, não basta atribuir a cada fila de entrada uma taxa máxima. Para exemplificar essa situação, considere que a recepção de tráfego de um enlace de acesso esteja sendo completamente utilizado por uma VM A. Caso a VM B, que compartilha os recursos da mesma máquina física de A, esteja recebendo menos tráfego do que lhe foi garantido, não é possível determinar se B está recebendo menos tráfego porque não existe demanda suficiente por parte de B ou se isso ocorre porque existe demanda por parte de B mas o enlace está completamente saturado pelo tráfego de A.

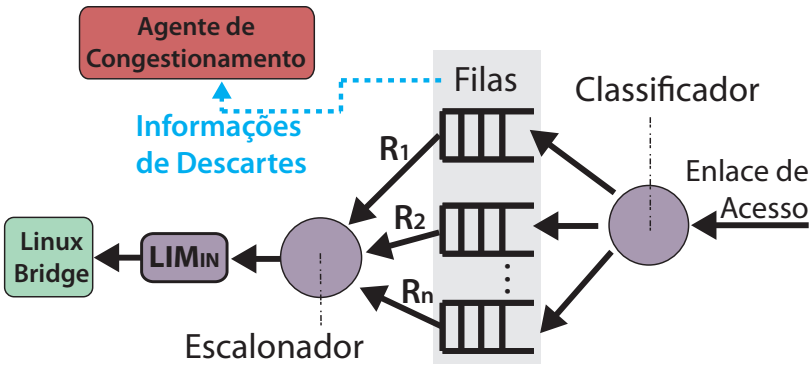


Figura 2.14. Escalonador de Entrada do Gatekeeper

Para determinar se o tráfego para cada VM extrapola os limites do sistema, o Gatekeeper transfere o gargalo existente no enlace de acesso para o próprio escalonador de entrada. Conforme mostrado na figura 2.14, o sistema distribui os pacotes que chegam pelo enlace de acesso em filas associadas às VMs de acordo com endereço MAC de destino. Quando as filas não estão vazias, o escalonador de entrada utiliza o mesmo algoritmo que o escalonador de saída para entregar os pacotes às VMs, garantindo as suas respectivas taxas de recebimento R_i. No entanto, o escalonador de entrada restringe a taxa agregada de entrega de pacotes às VMs em LIM_{IN}, uma taxa ligeiramente inferior à capacidade do enlace de acesso. Essa limitação cria uma faixa de banda reservada (*headroom*), equivalente à diferença entre a capacidade do enlace

e LIM_{IN}, permite ao sistema detectar situações de contenção seriam ofuscadas pelas limitações da rede de acesso.

Com a inserção do limite de entrada LIM_{IN}, a alocação de tráfego no sistema é feita de forma a nunca ultrapassar esse limite. Quando a taxa de recebimento de pacotes exceder LIM_{IN}, algumas filas começarão a acumular pacotes e eventualmente descartá-los quando a quantidade de pacotes a serem armazenados exceder a capacidade das filas. Como o escalonador de entrada remove os pacotes de cada fila a uma taxa garantida de pelo menos R_i, somente as filas que estão recebendo pacotes a uma taxa maior que a sua taxa garantida R_i irão exceder a sua capacidade e descartar pacotes. Esses descartes são utilizados pelo Gatekeeper como um indicador de excessos e como um mecanismo de controle.

Para garantir o escalonamento do tráfego de entrada com conservação de trabalho, não basta apenas satisfazer todas as garantias de banda. Também é preciso ser capaz de realocar a banda não utilizada para as VMs com demandas excedentes às suas garantias de banda. Para isso, ambos os escalonadores, de entrada e de saída, utilizam disciplinas que garantem valores mínimos de alocação, mas que são capazes de distribuir banda excedente entre os fluxos ativos (operação denominada *bandwith stealing* ou *borrowing* em diferentes contextos). Por esse motivo, no escalonador de entrada, se um fluxo excede a sua alocação mas existe banda disponível, aquele fluxo será servido a uma taxa mais alta que seu limite garantido R_i e não haverá descartes desnecessários.

2.4.2 Mecanismo de Controle de Alocação de Banda

A partir do momento o agente de congestionamento começa a ser notificado de descartes nos fluxos de entrada, o sistema assume que há um desequilíbrio na alocação de banda entre os clientes. Em um primeiro momento, ao invés de gerenciar explicitamente a alocação do tráfego de entrada para cada VM, o Gatekeeper opta por explorar as propriedades de autoadaptação de protocolos considerados "TCP-friendly", que implementam um controle de congestionamento semelhante ao de TCP, descrito na seção 2.2.1. Assim, tráfegos que possuem um bom controle de congestionamento deverão reagir ao descarte de pacotes buscando satisfazer a taxa de recebimento imposta implicitamente pelo Gatekeeper.

Enquanto o descarte seletivo de pacotes faz com que o tráfego "TCP-friendly" se adapte à taxa de recebimento imposta implicitamente, o tráfego considerado não-responsivo[5] pode não se adaptar corretamente à taxa imposta utilizando apenas o

[5]Adaptação do termo em inglês *non-responsive*, encontrado em diversos casos na literatura brasileira da área de Redes de Computadores.

descarte de pacotes. Para lidar com tráfego não-responsivo, o Gatekeeper emprega um mecanismo complementar ao descarte seletivo de pacotes acumulados nas filas de entrada. O agente de congestionamento é responsável por monitorar periodicamente a taxa de descarte de pacotes de cada uma das filas do escalonador de entrada. Caso esses descartes ultrapassem um certo limiar, o agente marca aquele fluxo como não responsivo.

O agente de congestionamento é responsável por enviar uma mensagem de notificação quando a taxa de descartes ultrapassa um limite preestabelecido D. Essa mensagem de notificação é enviada ao agente de congestionamento da máquina física originadora do fluxo não-responsivo. Como existe a possibilidade de que várias VMs estejam envolvidas com o tráfego não-responsivo, é necessário adotar uma política para selecionar qual dessas VMs irá receber uma mensagem de notificação. Na atual implementação do sistema, a mensagem de notificação é destinada à VM que enviou o ultimo pacote descartado pela fila de entrada. O sistema então impõe um novo limite, mais baixo que o atual, para a taxa de transmissão daquela VM (LIM_j na figura 2.13) utilizando um protocolo distribuído.

Para determinar qual agente de congestionamento deve receber a mensagem de notificação, o Gatekeeper mantém um serviço de diretório que mapeia os endereços das VMs para o endereço da máquina física na qual elas estão sendo executadas. Utilizando esse serviço é possível determinar o agente que deve receber a mensagem de notificação utilizando apenas o endereço da VM que está enviando o tráfego não responsivo.

Inicialmente, o escalonador de saída configura $LIM_{\{1..j\}}$ com o valor da largura de banda total do enlace de acesso, não restringindo qualquer tráfego até que uma mensagem de notificação seja recebida. O mecanismo de controle então segue um princípio semelhante àquele adotado por TCP. Assim que uma mensagem de notificação é recebida pelo agente de congestionamento, ele reduz pela metade o valor LIM_j correspondente à VM identificada na mensagem. Posteriormente, se novas mensagens de notificação não são recebidas, LIM_j é incrementado linearmente até que ele recupere seu valor inicial ou até que uma outra mensagem de notificação seja recebida.

A figura 2.15 exemplifica esse mecanismo utilizando duas VMs que enviam tráfego para um mesmo enlace de acesso. Esse exemplo assume que as duas VMs têm garantias iguais e, portanto, em caso de contenção, as duas VMs devem continuar consumindo recursos de rede de forma igual. As linhas contínuas representam o valor LIM_j de cada VM ao longo do tempo e a linha tracejada representa a alocação de banda ideal para cada uma das VMs de acordo com as demandas encontradas no enlace de acesso compartilhado. O limite ideal varia ao longo do tempo, de acordo com a quantidade de clientes que compartilham o mesmo enlace de acesso e da demanda pelos recursos de

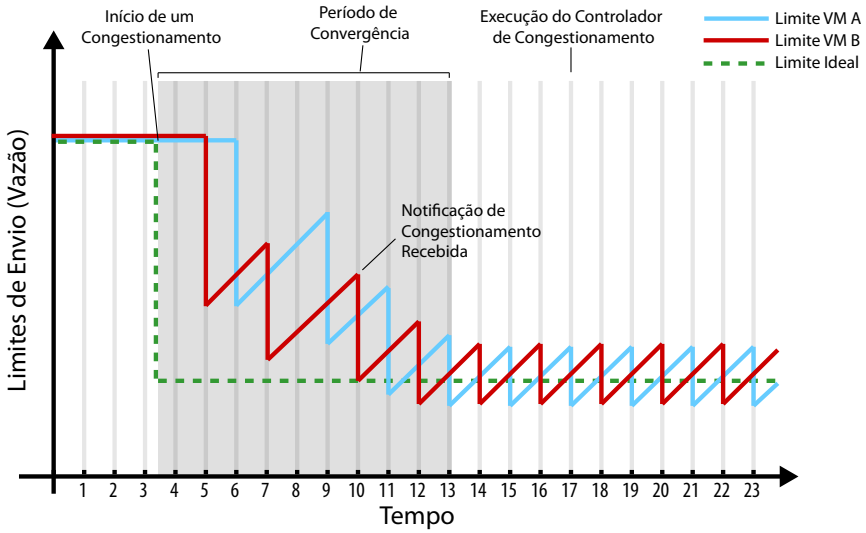


Figura 2.15. Comportamento dos Controladores de Transmissão do Gatekeeper quando ajustados utilizando mensagens de notificação enviadas por outros Agentes de Congestionamento

rede de cada um. Nesse exemplo assume-se que um outro cliente (cujos limites de suas VMs não são apresentados na figura) não apresentava qualquer demanda por recursos de rede até o tempo 3.5, quando o mesmo passou a utilizar sua banda garantida ocasionando na queda do limite ideal para as VMs A e B. As linhas verticais representam a periodicidade com que o Agente de Congestionamento realiza as medições de banda e envia mensagens de notificação.

2.4.3 Desempenho

Na implementação inicial do sistema [54, 55], os escalonadores de entrada e de saída utilizam os algoritmos de escalonamento de pacotes providos pelo Kernel Linux e o agente de congestionamento foi implementado como um programa no nível de usuário. Esse último componente é o responsável por quase todas as funções do Gatekeeper: realizar as leituras dos dados das filas de entrada e processá-los, enviar mensagens de notificação aos agentes de congestionamento instalados em outras máquinas físicas e ajustar os limites de banda de cada fila do escalonador de saída, tanto no recebimento de mensagens de notificação quanto durante a retomada de banda após um ajuste proveniente de uma mensagem de notificação. Os escalonadores de entrada e saída utilizam disciplinas de fila (*queuing Disciplines*) HTB do Linux (apresentadas na seção 2.2.3).

Para ter uma visão geral do desempenho do controle de tráfego realizado pelo

Gatekeeper, as figuras 2.17 e 2.18 apresentam os resultados de um dos experimentos realizados utilizando o protótipo descrito. Os experimentos envolvem dois clientes A e B com máquinas virtuais distribuídas em 5 máquinas físicas do ambiente de testes, conforme a figura 2.16. A capacidade de todos os enlaces é de $1Gbps$. As garantias de banda para os clientes A e B foram configuradas como $600Mbps$ e 43% desse valor, $260Mbps$, respectivamente. O experimento envolve a indução de congestionamento no enlace de acesso do servidor 1, que é compartilhado pelos dois clientes.

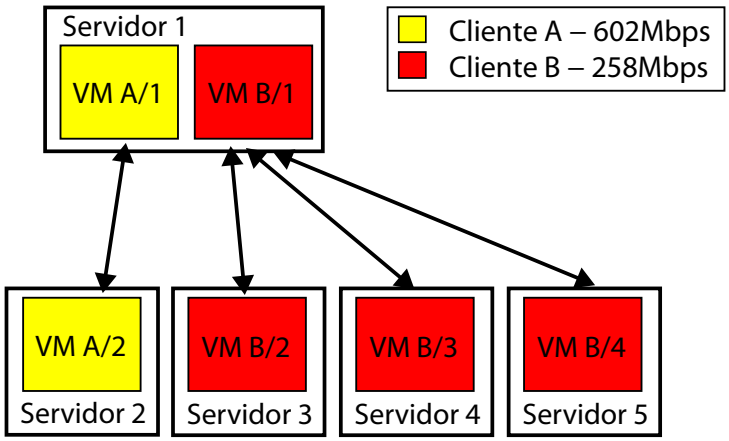


Figura 2.16. Alocação das máquinas virtuais e padrões de tráfego gerados no experimento de avaliação de desempenho do Gatekeeper.

A vazão de tráfego de cada VM foi medida para verificar a divisão de banda entre os dois clientes. As vazões observadas para A e B são apresentadas como barras empilhadas, sendo a inferior (amarela, ou cinza mais claro) a vazão de A. As linhas horizontais indicam a altura que, idealmente, as barras de A e B deveriam atingir, respectivamente.

Os resultados produzidos pelo Gatekeeper foram comparados com outros três cenários. No primeiro, denominado "Sem Controle", nenhuma solução de controle de tráfego é utilizada. A divisão da banda disponível é o resultado unicamente dos mecanismos de controle de congestionamento de cada fluxo (ou da falta deles). Nos outros dois cenários, denominados "*Limite Fixo*" e "*Empréstimo*", apenas o controle de banda com filas HTB é utilizado. No primeiro, os parâmetros `rate` e `ceil` de cada fila recebem valores iguais, representando as garantias de clientes. Nesse caso, não há qualquer tentativa de se distribuir banda excedente entre os clientes; mesmo que algum cliente não consiga utilizar sua banda alocada, o outro não deveria ser capaz de utilizá-la. No segundo cenário, o valor `rate` é ajustado com os valores das garantias e o parâmetro `ceil` é ajustado com o mesmo valor do parâmetro LIM_{IN}, $860Mbps$ para esse experi-

mento. Nesse caso, o HTB tentará realocar qualquer banda não utilizada de forma a realizar um escalonamento com conservação de trabalho.

Em cada cenário, foram consideradas quatro combinações de tipos de fluxos: (*i*) cada VM origem usa uma conexão TCP para cada destino; (*ii*) as VMs de A usam uma conexão, enquanto as de B usam 10 conexões (para enfatizar o impacto do controle de congestionamento por conexão de TCP); (*iii*) A usa TCP e B usa UDP; (*iv*) ambos usam UDP. Os tipos de tráfego utilizados pelos clientes são indicados logo abaixo das barras, sendo o primeiro deles utilizado pelo cliente A e o segundo utilizado pelo cliente B.

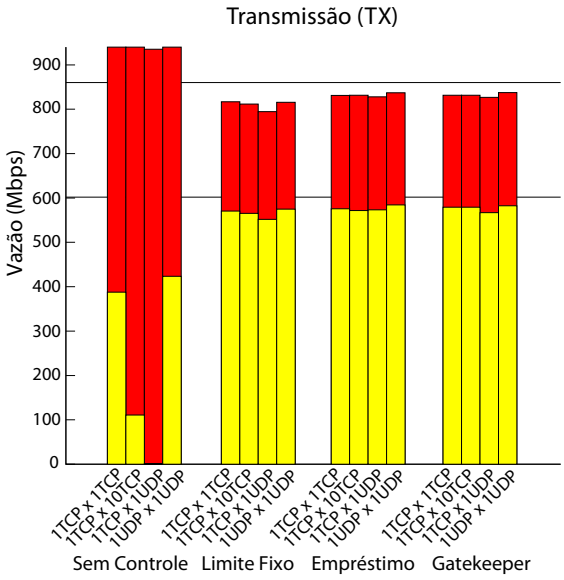


Figura 2.17. Resultados obtidos comparando o controle de tráfego do Gatekeeper com outras abordagens para o cenário TX.

A figura 2.17 apresenta os resultados da alocação de banda quando os fluxos gerados são *enviados* pelas VMs A/1 e B/1 (*TX*). Como esperado, a solução sem controle de alocação não permite obter uma alocação que seja próxima à desejada. Quanto às as soluções de controle de tráfego consideradas, todas são capazes de dividir a banda do enlace de acesso aproximadamente de acordo com as garantias de cada cliente para o cenário TX. Isso se deve ao fato do gargalo da comunicação ser o enlace ligado diretamente à máquina física que gerava todo o tráfego. Nesse caso, apenas o controle de banda no transmissor é suficiente para garantir a alocação definida. Como não há nenhuma banda excedente, as três soluções são equivalentes.

A boa divisão de tráfego, porém, não é mantida no cenário RX, quando os fluxos são *recebidos* pelas VMs A/1 e B/1. A figura 2.18 apresenta os resultados. Nesse caso

a contenção ocorre no núcleo da rede, quando três fluxos de B chegam ao *switch* de acesso ao servidor 1 junto com o fluxo de A. Como cada fluxo de B tem uma origem diferente, cada um inicia com 260 Mbps e o seu agregado extrapola a banda prevista para recepção pela máquina B/1, além de o tráfego total, considerando também o fluxo de A, ser superior à capacidade do enlace. Isso causa descartes no *switch* da rede, que não tem informação sobre as alocações, nem tem recursos para colocá-las em prática. A divisão de banda está de acordo com as garantias somente quando apenas fluxos TCP são utilizados, pois nesse caso a combinação do controle de congestionamento de TCP com os limitadores de entrada são suficientes para conseguir uma alocação próxima do ideal.

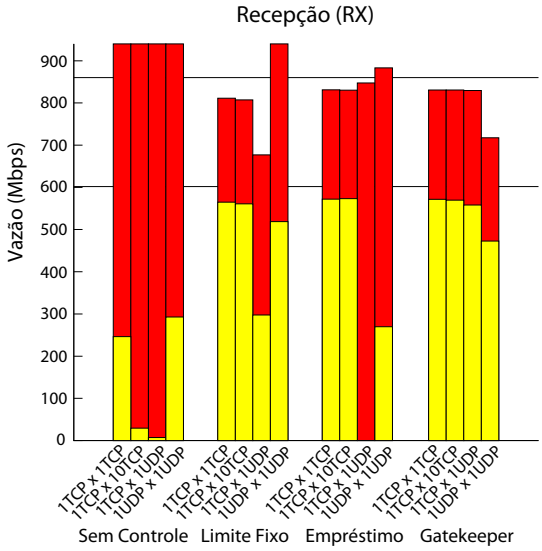


Figura 2.18. Resultados obtidos comparando o controle de tráfego do Gatekeeper com outras abordagens para o cenário RX.

Na presença tráfego não-responsivo, entretanto, o uso isolado do HTB não é capaz de controlar a alocação de banda de maneira correta. No cenário *Limite Fixo*, apesar da alocação de banda nos transmissores garantir algum recurso para TCP, as perdas ainda no *switch* ainda são superiores ao desejado e o fluxo TCP reduz sua taxa de transmissão abaixo do que lhe seria garantido. O os fluxos de B nesse caso conseguem utilizar mais banda que a quantidade que deveria ser alocada a eles (já que o limite é fixo), devido ao impacto do seu agregado. A solução com empréstimo, que permite que banda ociosa seja utilizada por outros fluxos, apresenta resultados ainda piores. Nesse caso, quando TCP inicialmente reduz a sua taxa de transmissão devido a perdas, o mecanismo de empréstimo identifica banda cedida por TCP como ociosa e a assinala para o fluxo UDP. Uma vez isso tendo ocorrido, TCP continua sofrendo perdas e reduzindo sua

taxa, nunca sendo capaz de recuperar a banda que foi erroneamente "emprestada" ao fluxo UDP.

O uso do mecanismo de notificação do Gatekeeper é capaz de oferecer uma melhor divisão de banda de acordo com as garantias de tráfego na presença de fluxos não responsivos. No caso em que o fluxo com mais recursos é TCP, o comportamento é praticamente igual aos casos apenas com TCP, bem próximo ao desejado. No caso com fluxos UDP apenas, o resultado com Gatekeeper foi visivelmente mais próximo do desejado que com qualquer das outras soluções, apesar de ainda não estar exatamente no nível esperado. É possível notar, a partir dos resultados apresentados, que o controle de tráfego do Gatekeeper pode ser melhorado.

2.4.4 Limitações

Os resultados apresentados indicam que o desempenho obtido com o Gatekeeper original ainda pode ser melhorado. Em situações de grande contenção, os mecanismos automáticos de redução multiplicativa e incremento aditivo não permitem uma estabilização rápida da alocação nos patamares desejados, como no caso com fluxos UDP do experimento anterior.

Além disso, a versão original do sistema, por falta de uma forma eficiente e estável para medição da vazão de cada fluxo presente em uma interface, depende exclusivamente da informação sobre descartes para operar. Uma análise posterior do sistema identificou dois inconvenientes nesse comportamento: primeiro, os descartes são afetados pelo tamanho das filas utilizadas no escalonador de entrada e pela margem definida entre a capacidade do enlace e o limite do escalonador, LIM_{IN} ; segundo, a identificação de fluxos não responsivos depende da observação do último pacote descartado.

A escolha do tamanho das filas e do limite de alocação de banda no escalonador de entrada são paâmetros do sistema e devem ser sintonizados experimentalmente. Isso foi feito através de experimentos com diversos padrões de fluxo, em busca de um valor que fosse aceitável [54, 55].

O elemento comutador original do Xen, a *linux bridge*, não oferece nenhum recurso de identificação de fluxos, nem pontos de monitoração simples para observar o estado das filas do mecanismo de controle de tráfego do Linux. Para sanar esse problema, a implementação do Gatekeeper original incluía uma alteração no código das *qdiscs* do Linux, a fim de recuperar o último pacote descartado por uma fila. Pela estrutura do *kernel*, não era possível reagir a cada descarte, apenas armazenar o último descarte, que poderia ser inspecionado pelo agente de congestionamento. Dessa forma, a identificação do "ultimo descarte" depende do escalonamento do processo agente e da

qdisc específica, ambos fora do controle do programador. A premissa nesse caso é que um fluxo não-responsivo, por sua natureza, será alvo de descartes com mais frequência que outros dentro de seus limites e, portanto, a probabilidade de um pacote desse fluxo ser observado é maior.

Finalmente, por essas limitações sobre o nível de informação disponível para o agente de congestionamento na interface do sistema operacional, quase toda a lógica de controle do sistema foi implementada a nível de usuário. Dessa forma, a monitoração de descartes requer transições frequentes da fronteira do *kernel*, criando um *overhead* de processamento para o sistema.

Por esses motivos, uma nova implementação para o sistema era desejável. Uma nova solução se tornou mais interessante quando o *Open vSwitch* foi identificado como uma ferramenta útil nesse processo. Como discutido na seção a seguir, esse sistema oferece uma interface muito mais rica para a monitoração de fluxos no elemento de comutação, o que se mostrou um elemento importante para o desenvolvimento de uma nova solução.

2.5 O arcabouço *Open vSwitch*

Open vSwitch é uma implementação de um elemento de chaveamento de pacotes que implementa a arquitetura *OpenFlow*. Desde seu lançamento ela tem sido adotada em ambientes virtualizados como uma implementação eficiente e flexível para um *switch* virtual. Essa larga adoção, sua estrutura modular e alguns dos recursos da arquitetura *OpenFlow* motivaram sua adoção como arcabouço para a implementação da nova versão do Gatekeeper.

2.5.1 OpenFlow

A arquitetura *OpenFlow* foi proposta como uma forma de viabilizar pesquisa e extensão no ambiente de redes sem inviabilizar a utilização das mesmas para atividades "de produção" [37]. Uma característica básica da arquitetura é uma separação clara entre os planos de dados (*data plane*) e controle (*control plane*) em elementos de chaveamento.

O plano de dados cuida do encaminhamento de pacotes com base em regras simples associadas a cada entrada da tabela de encaminhamento do comutador de pacotes (um *switch* ou roteador). Essas regras, definidas pela arquitetura, incluem encaminhar o pacote como recebido ou após reescrever parte de seus cabeçalhos, descartá-lo, ou encaminhá-lo para inspeção por um controlador da rede. Em dispositivos dedicados, esse plano pode ser implementado em hardware utilizando os elementos comuns a ro-

teadores e *switches* atuais. Já o módulo de controle permite ao controlador da rede programar as entradas dessa tabela de encaminhamento com padrões que identifiquem fluxos de interesse e as regras associadas a eles. O elemento controlador, pode ser um módulo de software implementado de forma independente em algum ponto da rede.

Um grande trunfo da arquitetura *OpenFlow* é a flexibilidade que ela oferece para se programar de forma independente o tratamento de cada fluxo observado, do ponto de vista de como o mesmo deve (ou não) ser encaminhado pela rede. Esse controle é representado através de entradas na tabela de fluxos da arquitetura.

Um fluxo é definido como um padrão de bits representado em uma memória TCAM (*Ternary Content-Addressable Memory*). Nesse tipo de memória bits podem ser representados como zero, um ou "não importa" (*don't care*), indicando que ambos os valores são aceitáveis naquela posição. Como o padrão é programado a partir do plano de controle, fluxos podem ser definidos da forma escolhida pelo controlador (p.ex., todos os pacotes enviados a partir do endereço físico A para o endereço físico B, ou todos os pacotes TCP enviados da máquina com endereço IP X para o porto 80 da máquina com endereço IP Y). Cada pacote que chega a um comutador *OpenFlow* é comparado com cada entrada dessa tabela; caso um casamento seja encontrado, considera-se que o pacote pertence àquele fluxo e aplica-se as regras definidas pelo controlador. Caso um casamento não seja encontrado, o pacote é encaminhado para o controlador para ser processado — o que pode resultar na criação de uma nova entrada para aquele fluxo. Além das regras, a arquitetura prevê a manutenção de três contadores por fluxo: pacotes e bytes trafegados e duração do fluxo.

2.5.2 Open vSwitch

A implementação de referência do modelo *OpenFlow* é um comutador em software, executando no espaço de usuário em uma máquina Linux. Esse modelo tem sido utilizado como base em diversos experimentos, mas carece de um melhor desempenho.

Desenvolvido para suprir essa lacuna, o *Open vSwitch* (OvS) é um *switch* virtual que segue a arquitetura *OpenFlow*, implementado em software, com o plano de dados dentro do *kernel* do sistema operacional Linux, enquanto o plano de controle é acessado a partir do espaço de usuário. Em particular, essa implementação foi desenvolvida especificamente para controlar o tráfego de rede da camada de virtualização em ambientes virtualizados.

Em sua implementação atual, o *Open vSwitch* é composto por dois componentes principais: um módulo presente no núcleo do sistema operacional, denominado "*Fast Path*", e um componente no nível de usuário, o "*Slow Path*". O *fast path* interage

ativamente com o tráfego de rede, pois é responsável por procurar rotas na tabela de fluxos e encaminhar pacotes de rede. Já o *slow path* é o componente do *Open vSwitch* onde são implementadas as demais funcionalidades associadas ao plano de controle, como as interfaces de configuração do switch, a lógica de uma ponte com aprendizado (*learning bridge*) e as funcionalidades de gerência remota, etc. A interação entre os dois módulos se dá prioritariamente através da manipulação da tabela de fluxos.

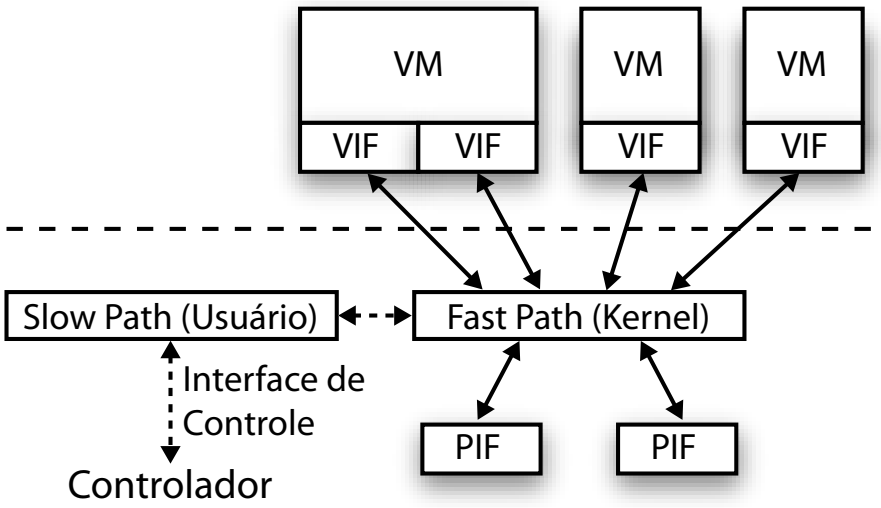


Figura 2.19. Arquitetura do Open vSwitch

A implementação do *fast path* é simples e seu código é composto por poucas linhas (em torno de 3000 — um número de linhas bem pequeno quando comparado às 30.000 do *slow path*). Existem dois motivos para essa decisão de projeto, sendo que o primeiro deles é o desempenho. O *fast path* é a parte crítica do sistema quando consideramos o tempo de processamento e, portanto, quanto menor o processamento realizado para cada pacote, maior a capacidade de processamento desse componente. Essa característica torna indispensável a sua implementação como uma parte do sistema operacional, posicionando-o mais próximo às NICs. Além disso, como a lógica implementada pelo *fast path* é dependente das APIs do sistema operacional, manter a sua complexidade pequena facilita a tarefa de portar o *Open vSwitch* a um outro sistema operacional. O segundo motivo é o esforço reduzido em adaptar as funcionalidades do *fast path* à aceleração de hardware (*hardware acceleration/offloading*), que eventualmente pode ser oferecida pela máquina física.

Além de oferecer as funcionalidades da arquitetura *OpenFlow*, o OvS, na sua configuração padrão, opera como um *switch* Ethernet normal. Para simplificar a sua integração com o *kernel* Linux, o OvS emula as interfaces de rede daquele sistema e

utiliza partes do código fonte de seu módulo *bridge*, que implementa o *switch* de rede padrão do Linux. Como resultado dessas decisões de projeto, o *Open vSwitch* pode ser utilizado como um substituto imediato para os *switches* virtuais adotados pelos VMMs baseadas em Linux mais utilizados atualmente, como Xen, XenServer e KVM e vem sendo adotado como o *switch* padrão em diversas distribuições, mesmo quando as funcionalidades da arquitetura *OpenFlow* não são utilizadas.

2.6 Trabalhos Relacionados

As soluções existentes para o controle de tráfego em ambientes virtualizados, presentes em NIC modernas, sistemas operacionais e *hypervisors*, simplesmente impõem um limite máximo para as taxas de envio e recebimento de dados de cada máquina virtual. Soluções que utilizam apenas limites fixos estão propensos a desperdiçar recursos de rede mesmo que haja demanda para o consumo desses recursos. Essas soluções não são capazes de prover o controle de tráfego respeitando as garantias exigidas por cada cliente oferendo um escalonamento com conservação de trabalho, assim como o Gatekeeper.

Recentemente foram propostas várias outras soluções de controle de tráfego para datacenters [23, 31, 52, 56]. Esta seção apresenta algumas dessas soluções, dando maior ênfase e discutindo com mais detalhes as soluções que apresentam maior grau de semelhança comparado ao Gatekeeper. De um modo geral, essas soluções não são capazes de oferecer um controle de tráfego tão eficiente e/ou previsível como o Gatekeeper, mesmo aquelas com alto grau de semelhança , como é o caso do Seawall [52, 53].

O Seawall é uma solução de controle de tráfego centrada em servidores, baseada em notificações e que realiza o controle de tráfego utilizando limitadores de banda em cada máquina física. É portanto uma das soluções para controle de tráfego em ambientes visualizados que mais se assemelha ao Gatekeeper. A principal ideia do Seawall é tornar o VMM responsável por auxiliar no controle de congestionamento do TCP, ao invés de deixar essa tarefa somente sob o controle das VMs. Para isso, o tratamento de pacotes realizado pelo VMM deve ser alterado para adicionar contadores aos cabeçalhos dos pacotes transmitidos por cada VM. Esse contadores são utilizados apenas pelos VMMs para monitorar o estado da rede. Quando um pacote chega ao destino, o VMM retira esses contadores e os analisa antes de entregar o pacote à VM. Como os contadores são estritamente crescentes, o VMM que recebe os pacotes consegue calcular quantos pacotes foram perdidos na rede e então avisar ao VMM que hospeda

a VM transmissora se houve ou não perda de pacotes. O lado transmissor pode então decidir por limitar o envio de tráfego da VM de acordo com os pesos definidos para cada fluxo.

O Seawall garante que, para todos os enlaces da rede, a divisão dos recursos daquele enlace será feita de acordo com o peso atribuído para cada fluxo. O problema com essa abordagem é que quanto maior o número de fluxos distintos utilizando um mesmo enlace da rede, menor será a parcela daquele enlace atribuída a cada fluxo. Como a atribuição de recursos para cada fluxo é feita com base no enlace com menor quantidade de recursos disponíveis, todo tráfego de rede que atravessa esse enlace é limitado pelo Seawall. Essas características tornam a atribuição de recursos de cada cliente imprevisível, podendo variar muito dependendo dos padrões de tráfego presentes na rede.

O AF-QCN [31] é uma extensão para o QCN[6] [2], que implementa o controle de tráfego centrado na rede com auxílio de limitadores de banda instalados nos servidores. Assim como o Seawall, ele também é baseado em notificações e realiza o controle de tráfego utilizando limitadores de banda, porém esses são implementados no hardware de cada NIC, ao invés de serem implementados em software dentro do VMM. O motivo para isso é que AF-QCN não é uma solução destinada à ambientes virtualizados, embora também possa ser utilizada para realizar o controle de tráfego de VMs.

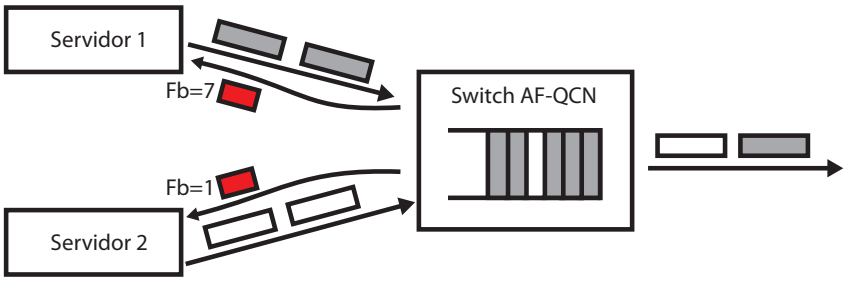


Figura 2.20. Visão geral do algoritmo AF-QCN. Mensagens de notificação enviadas pelos *switches* às NICs estão representadas em Vermelho

O controle de tráfego realizado pelo AF-QCN é baseado na monitoração do da fila de saída de cada interface dos switches da rede. Quando a quantidade de pacotes na fila atinge um certo limiar, o switch assume que um congestionamento está para ocorrer e passa a coletar pacotes da fila para enviar mensagens de notificação aos servidores identificados como origem desses pacotes. A mensagem contém um valor Fb calculado pelo switch que descreve a severidade do congestionamento. Esse valor é utilizado pelos servidores para ajustar a taxa de envio dos limitadores de tráfego. A

[6] O QCN é atualmente um padrão definido pela IEEE, o 802.1Qau

frequência de colecta de pacotes também muda de acordo com o valor calculado para Fb. No QCN, esse valor é o mesmo para cada pacote colectado. O AF-QCN propõe uma diferenciação do falir Fb para cada fluxo, baseado no peso atribuído para cada fluxo. Com isso é possível que o switch envie diferentes valores Fb para cada servidor, baseado na prioridade de cada um dos fluxos.

Assim como o Seawall, o AF-QCN não oferece um controle de tráfego previsível, sendo a quantidade de recursos de rede atribuídas a cada cliente dependente dos padrões de tráfego observados nos switches. Além disso, a solução depende de que o hardware dos switches seja capaz de armazenar o consumo de cada fluxo, o que pode ser proibitivo em termos de escalabilidade.

Um outro trabalho de pesquisa que provê controle de tráfego com garantias de banda é o Secondnet [23]. A proposta do Secondnet é mais ampla que a do Gatekeeper, pois além de controlar a largura de banda, ele também descreve a forma com que o encaminhamento de pacotes deve ser feito na rede do Datacenter. Um entidade central, chamada de gerenciador de datacenter virtuais (*Virtual Datacenter (VDC) Manager* é responsável por distribuir as VMs nas máquinas físicas de acordo com as garantias de tráfego solicitadas por cada cliente. Com isso, o sistema é capaz de prover garantias de banda definidas para cada par de VMs que são monitoradas e gerenciadas por pelo VDC Manager. A ideia é que cada cliente possua seu próprio datacenter virtual (VDC), sendo a infraestrutura do Secondnet responsável por garantir o isolamento de tráfego entre diferentes VDCs.

Enquanto o Secondnet é capaz de prover garantias de banda para todo par de VMs, a forma com que as garantias devem ser descritas pelo usuário não é intuitiva. Em geral, os clientes de um datacenter não conhecem os padrões de comunicação de suas aplicações bem o suficiente para determinar qual deve ser a garantia de tráfego entre cada par de máquinas virtuais. Além disso, os padrões de tráfego podem ser dinâmicos, variando dinamicamente ao longo do tempo. Ao criar reservas de recursos de rede entre cada par de VM pode levar a uma baixa utilização média da rede, pois é provável que não exista uma demanda constante por esses recursos. O algoritmo de alocação de VMs centralizado também é um potencial problema do sistema em termos de escalabilidade.

Até onde sabemos, a única solução de controle de tráfego para datacenters virtualizados que é capaz de de prover isolamento de recursos de rede com garantias de tráfego definidas por clientes, ao invés de definidas por fluxos, é o Netshare[34]. No entanto, o Netshare também depende de um controlador central, assim como o Secondnet. Isso limita a escalabilidade dessas soluções, que podem não ser capazes de lidar com as mudanças repentinas dos padrões de tráfego e com as frequentes mudanças

de máquinas virtuais e clientes encontradas em datacenters que oferecem recursos no modelo de computação em nuvem.

Um outro trabalho de pesquisa relacionado ao Gatekeeper é o *Core-Stateless Fair Queuing (CSFQ) para a Internet* [56]. A proposta desse trabalho é manter as informações sobre o tráfego nos roteadores de borda que estão ligados aos enlaces de acesso. Esses roteadores são responsáveis por classificar e adicionar um identificador a cada pacote enviado à rede. Nesse cenário, é o núcleo da rede que provê garantias de tráfego através do descarte seletivo de pacotes de acordo com os identificadores adicionados pelos roteadores de borda. O CSFQ assume que os servidores são não-confiáveis, assim como o Gatekeeper, porém concentra toda a tarefa de controle de tráfego no núcleo da rede. O motivo para essa decisão de projeto é o ambiente para o qual o trabalho foi desenvolvido, a Internet, onde não existe uma camada de software que pode ser gerenciada pelo ISP. Em contraste, o Gatekeeper se concentra em um ambiente virtualizado gerenciado por um único administrador, não precisando contar com o apoio de *switches* ou roteadores da rede.

O Gatekeeper é capaz de previvir ataques de negação de serviço distribuídos (DDoS attacks) e apresenta grandes diferenças quando comparado a outros mecanismos que oferecem qualidade de serviço utilizando apenas roteadores [27]. Pode-se dizer que o Gatekeeper é complementar ao *Cloud Control* [47]. O Cloud Control utiliza um protocolo distribuído complexo para realizar o controle de tráfego entre datacenters espalhados geograficamente. O Gatekeeper foca em prover o controle de tráfego com garantias para clientes que utilizam os recursos de um mesmo datacenter e utiliza uma abordagem muito mais simples que a utilizada pelo Cloud Control para atingir os seus objetivos.

Existem também outras soluções que propõem realizar o controle de toda a infra-estrutura computacional, assim como o Secondnet [25, 44, 57, 58, 61, 68, 70]. Porém, essas soluções se preocupam apenas com a alocação de VMs em termos de memória e CPU, não oferecendo isolamento de tráfego e nem garantias do consumo de recursos de rede às VMs.

Capítulo 3

Gatekeeper-ng

A proposta original do Gatekeeper deixou várias oportunidades para melhorias. Nesse trabalho, algumas dessas oportunidades foram exploradas com o objetivo de aprimorar o controle de tráfego oferecido pelo sistema, resultando em nova versão do Gatekeeper, aqui referenciada como Gatekeeper-ng. A nova versão conta com uma nova arquitetura e novos algoritmos, desenvolvidos para contornar alguns dos principais pontos fracos da versão original do sistema, como o controle de tráfego ineficiente frente a um grande número de fluxos e/ou clientes e a grande quantidade de descartes de pacotes necessários para ajustar a alocação dos recursos quando ocorrem congestionamentos. As seções seguintes contém os detalhes referentes à arquitetura do novo sistema, aos algoritmos e aos detalhes implementação.

O Gatekeeper-ng mantém a ideia original do Gatekeeper, que é ajustar a taxa de envio dos transmissores para suprimir os congestionamentos, porém o modo como o sistema realiza essa tarefa difere significativamente entre as duas versões. A nova arquitetura e os novos algoritmos para definir as alocações de banda foram desenvolvidos para utilizar de forma mais eficiente as informações a respeito dos padrões de tráfego de rede que podem ser coletadas por cada máquina física utilizando um elemento de chaveamento de última geração como o *Open vSwitch*. Como resultado, os experimentos realizados (apresentados no capítulo 4) mostram que a nova versão, quando comparada à anterior, apresenta maior estabilidade e melhor precisão no controle de tráfego.

O desenvolvimento do Gatekeeper-ng foi planejado buscando melhorar a eficiência do sistema anterior, tanto em termos de custo computacional, objetivando a redução do consumo de CPU, quanto na eficiência do consumo de recursos, buscando um melhor aproveitamento da infraestrutura de rede. Quanto ao isolamento de tráfego, que é a principal função do Gatekeeper, o novo algoritmo de alocação de banda é capaz de determinar a divisão ótima da banda disponível no enlace de acesso quando se

depara com um congestionamento. Com isso, os limites de envio das máquinas virtuais envolvidas no congestionamento podem ser ajustados instantaneamente para um valor exato, eliminando o *período de convergência* existente na versão anterior do Gatekeeper, conforme apresentado na figura 2.15. Essas melhorias são obtidas utilizando algoritmos simples e de baixa complexidade computacional.

O Gatekeeper-ng mantém o modelo de garantias utilizado pela versão original do sistema, no qual os clientes possuem a visão lógica de que suas máquinas virtuais estão ligadas a um único *switch* lógico não bloqueante (*non-blocking*). O sistema controla a alocação de banda de cada cliente em função das garantias de largura de banda definidas para o enlace de acesso de cada máquina virtual. A vantagem desse modelo de garantias é a definição das políticas de consumo de forma simples e intuitiva, tanto para os operadores da rede quanto para os clientes do provedor de serviços. Como um exemplo prático da aplicabilidade do modelo, um provedor de recursos poderia definir diferentes classes de consumo de recursos com diferentes garantias de banda, deixando seus clientes livres para decidir qual classe de consumo desejam contratar de acordo com o volume de tráfego gerado por suas aplicações.

A apresentação do Gatekeeper-ng é organizada nas três seções a seguir. A primeira apresenta a nova arquitetura e o modo como é feita a coleta de informações a respeito dos congestionamentos, que serão necessárias para determinar a alocação de tráfego de cada cliente. A seção seguinte discute os algoritmos utilizados pelo sistema e as vantagens em comparação aos algoritmos adotados na versão anterior. A terceira seção apresenta uma descrição detalhada da implementação do Gatekeeper-ng no ambiente Linux, finalizando a descrição da solução proposta.

3.1 Arquitetura

A principal diferença relacionada à arquitetura do Gatekeeper-ng, quando comparado ao Gatekeeper original, foi a remoção das filas de entrada devido à mudança do mecanismo de detecção de congestionamentos.

Na versão anterior do Gatekeeper, as filas de entrada eram utilizadas para classificar o tráfego de cada cliente e assim obter informações referentes às demandas de cada um. O controlador de entrada era configurado com um limite agregado inferior à capacidade do enlace para que o gargalo fosse transferido dos *switches* de acesso para as máquinas físicas. Os descartes nas diversas filas serviam para identificar quais VMs excediam sua garantia de banda. Além disso, esses descartes também eram utilizados como um efeito colateral do comportamento inadequado de um fluxo. No entanto,

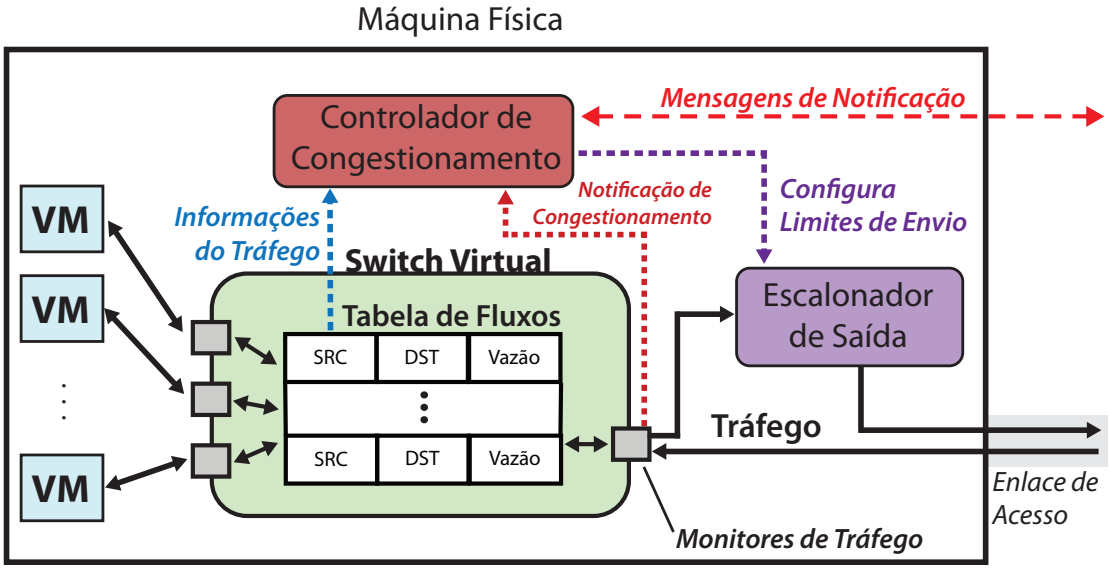


Figura 3.1. Arquitetura do Gatekeeper-ng

além das limitações operacionais discutidas na seção 2.4.4, essa estratégia não permite graduar precisamente quanto cada fluxo encaminhado para uma única fila de entrada excede sua alocação. Na tentativa de contornar essa limitação, o Gatekeeper assume que o fluxo que mais contribui para o congestionamento será também aquele que possui mais pacotes na fila do escalonador de entrada. Por esse motivo, o Gatekeeper utiliza o endereço de origem do último pacote descartado de cada fila como um indicador da VM que mais contribui para o congestionamento.

O Gatekeeper-ng adota uma outra solução, que é manter um controle direto do consumo de banda de cada fluxo observado na recepção de tráfego do enlace de acesso. Com isso o controlador de entrada foi substituído por outros dois componentes. O primeiro deles é um monitor de tráfego que observa a vazão da recepção de tráfego no enlace de acesso e o segundo é uma tabela que mantém a vazão observada para cada fluxo ativo em uma máquina física. Para ter acesso a todo o tráfego da máquina física, esses componentes devem ser implementados no *switch* virtual que interliga as máquinas virtuais e a interface externa. Uma visão geral da arquitetura do Gatekeeper-ng é apresentada na figura 3.1.

Cada entrada da tabela de fluxos possui informação suficiente para identificar cada fluxo de entrada na máquina física em termos da VM de origem e da VM de destino do tráfego. Além disso, cada entrada mantém um acompanhamento da vazão de cada fluxo, atualizada à medida que pacotes são processados pelo *switch* virtual. O monitor de tráfego realiza a mesma monitoração, porém referente à vazão agregada

de todos os fluxos que são recebidos pelo enlace de acesso. Apesar dessa informação poder ser derivada das entradas na tabela de fluxos, ela é computada separadamente por questões de desempenho.

A função do monitor de tráfego é detectar congestionamentos nos enlaces de acesso. Para isso, o Gatekeeper-ng adota uma ideia semelhante à utilizada pelo Gatekeeper original: manter a utilização do enlace ligeiramente abaixo de sua capacidade, no entanto os fluxos que infringem não têm seus pacotes descartados. Quando a vazão de tráfego observada pelo monitor ultrapassa um certo limiar, ligeiramente inferior à capacidade do enlace, isso indica que a demanda pela recepção de tráfego direcionado a uma (ou mais de uma) VM local excedeu a sua garantia de banda, podendo prejudicar as garantias de recepção de tráfego de outras VMs alocadas na mesma máquina física. Nessas ocasiões, o monitor de tráfego do Gatekeeper-ng assume que há uma violação no consumo de recursos do enlace de acesso e notifica o controlador de congestionamentos.

Ao ser informado do problema, o controlador utiliza os dados armazenados na tabela de fluxos para determinar qual é o consumo de banda exato de cada fluxo direcionado às VMs locais Como os fluxos estão relacionados às máquinas virtuais, essa informação, em conjunto com um serviço de diretório que mapeia máquinas virtuais aos respectivos clientes, pode ser traduzida para as demandas de cada cliente. Utilizando esses dois mecanismos é possível obter as informações necessárias para recalcular a alocação de banda de forma precisa, considerando as demandas de tráfego observadas e evitando o descarte de pacotes que já chegaram ao seu destino após atravessar toda a malha de rede consumindo recursos da infraestrutura.

Ao contrário das modificações no caminho de entrada do tráfego, o processamento do tráfego de saída saída ainda utiliza o mesmo escalonador de saída do sistema original. Cada uma das máquinas virtuais possui uma fila nesse escalonador e os pacotes dessas filas são consumidos de acordo com as garantias de tráfego associadas a cada máquina virtual.

3.2 Algoritmos

Devido à coleta de informações mais detalhadas a respeito dos congestionamentos e dos padrões de tráfego envolvidos no problema, o Gatekeeper-ng consegue realizar uma divisão de banda mais eficaz quando comparado à versão anterior do sistema. O novo algoritmo de alocação de banda foi projetado para maior tirar proveito dessas informações. Os detalhes dos algoritmos envolvidos são discutidos a seguir.

As estruturas de dados utilizadas pelo Gatekeeper-ng são relativamente simples.

Todo o funcionamento do sistema é baseado na largura de banda (vazão) medida pelo monitor de tráfego e pelas entradas da tabela de fluxos. Como a vazão é calculada pela quantidade de tráfego que atravessa o enlace em um intervalo de tempo fixo, a noção do tempo percorrido é fundamental para o funcionamento do Gatekeeper-ng. Qualquer fonte de tempo oferecida pela plataforma que executa o sistema, desde que apresente alta precisão, pode ser utilizada para calcular a vazão.

O algoritmo executado pelo monitor de tráfego é apresentado na figura 1. A cada pacote recebido, o monitor atualiza seu contador de dados e verifica, utilizando o relógio do sistema, se o intervalo de medição foi ultrapassado ou não. Quando o intervalo de medição é finalizado, o medidor de tráfego computa a largura de banda do intervalo e verifica se o limiar pré-definido foi ultrapassado, ou não. Caso isso tenha ocorrido (indicativo de congestionamento), é tarefa do monitor de tráfego notificar o controlador de congestionamento sobre o incidente. Na versão anterior do Gatekeeper, essa verificação era responsabilidade do controlador, que precisava consultar o escalonador de entrada continuamente para monitorar as perdas ocorridas.

Algoritmo 1: Algoritmo Executado por um Monitor de Tráfego

Entrada: Cada pacote PKT recebido pela Interface IF onde o Monitor MT está acoplado

```
1  início
2      MT.Contador ← MT.Contador + Tamanho(PKT);
3      se TempoAtual − MT.TempoUltimaMedicão > IntervaloDeMedicão
       então
4          MT.Vazão ← ( MT.Contador / (TempoAtual − MT.TempoUltimaMedicão) );
5          se MT.Vazão > IF.Capacidade − IF.Headroom então
6              Notifica congestionamento ao Controlador de Congestionamentos;
7          fim
8          MT.Contador ← 0;
9      fim
10 fim
```

O algoritmo 2, executado para atualizar a tabela de fluxos deve ser acoplado ao código que faz o processamento de pacotes do *switch* virtual. A estrutura de dados necessária para implementar a tabela de fluxos é uma tabela de espalhamento (tabela *hash*), onde o valor submetido à função de espalhamento é um identificador do fluxo (p.ex., uma combinação dos endereços fonte e destino do pacote a ser encaminhado

pelo *switch* virtual). O algoritmo é muito semelhante ao utilizado para atualizar as variáveis do monitor de tráfego. As diferenças estão na busca da entrada correspondente ao pacote processado utilizando uma função de espalhamento e na ausência do código para notificar o controlador de congestionamentos.

Algoritmo 2: Algoritmo Executado para Atualizar a Tabela de Fluxos

Entrada: Cada pacote PKT processado pelo *switch* virtual

```
1 início
      /* Código a ser inserido no Processamento do switch virtual
         */
2     IndiceT ← FuncãoDeEspalhamento(PKT.Fonte, PKT.Destino) ;
3     EntradaT ← TabelaDeFluxos[IndiceT];
4     EntradaT.Contador ← EntradaT.Contador + Tamanho(PKT);
5     se
      TempoAtual − EntradaT.TempoUltimaMedicão > IntervaloDeMedicão
      então
6         EntradaT.Vazão ← EntradaT.Contador/(TempoAtual −
          EntradaT.TempoUltimaMedicão) ;
7     fim
8     EntradaT.Contador ← 0;
9 fim
```

Os dois algoritmos discutidos anteriormente realizam a coleta das informações necessárias para o funcionamento do Gatekeeper-ng. Esses algoritmos são extremamente simples, com objetivos limitados a manter atualizados um conjunto de contadores que armazenam a vazão do tráfego observado em diferentes pontos da arquitetura. A simplicidade desses algoritmos tem um motivo: sua execução é realizada a cada pacote recebido pela máquina física e, como são executados muito frequentemente, é desejável que seus códigos sejam tão simples quanto possível.

Ao ser notificado da ocorrência de um congestionamento, o controlador de congestionamento deve tomar as devidas providencias para resolver o problema. Nesse instante, esse componente do sistema dá início à execução do Algoritmo 3, que sumariza as informações referentes ao congestionamento que deverão ser enviadas às máquinas físicas que hospedam as VMs responsáveis pelo problema.

Um dos problemas enfrentados ao realizar um controle de tráfego distribuído é a pouca (ou nenhuma) informação que transmissores e receptores possuem a respeito dos congestionamentos presentes na malha de rede. Ao detectar um congestionamento,

Algoritmo 3: Algoritmo de Alocação de Banda

Entrada: Notificação de Congestionamento pelo Monitor de Tráfego, Tabela de Fluxos TF e Mapa de VMs Locais M

início
  para *cada entrada* E *em* TF **faça**
    $M[E.VM\_Destino].Vazão \leftarrow M[E.VM\_Destino].Vazão + E.Vazão$
  fim
  `/* Buscando em` M `a máquina virtual local` VM_C`, que mais contribui para o congestionamento: */`
  $VM_C \leftarrow max_i(M[VM_i].Vazão - M[VM_i].Garantia)$;
  $N \leftarrow$ Número de VMs que enviam tráfego à VM_C;
  `/* Informação obtida a partir da Tabela de Fluxos */`
  `/* Calcule a largura de banda não utilizada` U `*/`
  $U \leftarrow \sum M[VM_i].Garantia \times B_i, \forall_i \in$ Conjunto de VMs Locais;
  `/* A variável binária` B_i `indica se a` VM_i `está utilizando a sua garantia de banda ou não */`
  Crie uma mensagem de notificação M contendo os valores N e U e envie uma cópia de M a cada máquina física que hospeda VMs enviando tráfego à VM_C;
fim

seja através de perdas de pacotes ou por um aumento da latência, os transmissores não têm a capacidade de determinar qual é a porção do enlace congestionado que eles poderiam utilizar de forma a acabar com o problema. Como consequência dessa falta de informação, não resta outra opção aos transmissores senão reduzir a taxa de transmissão de dados bruscamente na tentativa acabar com o congestionamento imediatamente. Após isso, eles iniciam uma retomada da taxa de envio lentamente, até seja detectado um outro congestionamento. Essa é a estratégia adotada pelo Protocolo TCP e também pelo Gatekeeper original.

Uma das maneiras de resolver o problema da falta de informação por parte dos transmissores é utilizar mecanismos baseados em reservas, como o IntServ [10]. Em um ambiente que adota essa solução, cada servidor é responsável por fazer uma solicitação de recursos da rede antes de iniciar a transmitir os seus dados. O IntServ especifica um protocolo para que os servidores possam fazer solicitações de reservas recursos junto aos elementos da rede. Dessa forma é possível evitar congestionamentos: caso um enlace de rede não seja capaz de suportar todo o tráfego a ser submetido a ele, os

elementos da malha de rede passarão a responder negativamente aos pedidos de reserva de recursos que envolvem o enlace de rede saturado. Apesar de resolver o problema, soluções baseadas em reservas não são viáveis para um ambiente de datacenter devido a sua baixa escalabilidade. Esses algoritmos envolvem o armazenamento de uma grande quantidade de variáveis de estado, referentes a cada fluxo que realiza uma reserva de recursos, para cada elemento de chaveamento da rede.

Nesse trabalho a ideia adotada pelo Gatekeeper para solucionar esse problema foi mantida: notificar os transmissores a respeito dos congestionamentos. A nova versão, porém, coleta informações mais detalhadas a respeito do consumo de recursos nos enlaces de acesso. Com isso o Gatekeeper-ng é capaz de enviar informações mais precisas aos transmissores a respeito da utilização do enlace congestionado. Essas informações são valiosas para os transmissores que estão causando o congestionamento, pois eles podem ajustar sua taxa de envio cientes da disponibilidade de recursos no enlace. Dessa forma é possível minimizar, ou até mesmo eliminar, o período de convergência para o limite ideal, discutido na seção 2.4. A tarefa do Algoritmo 3 é quantificar o uso do enlace de acesso e determinar as informações que serão enviadas aos transmissores que estão contribuindo para o congestionamento.

O primeiro passo do Algoritmo 3 é selecionar a máquina virtual local VM_C que está contribuindo de forma mais expressiva ao congestionamento (linha 5). A diferença entre a vazão de tráfego recebida e a vazão de tráfego garantida quantifica a violação das garantias de tráfego de uma VM. Após a ocorrência de um congestionamento e posterior seleção da VM_C, o Gatekeeper-ng é capaz de verificar também quais são as VMs que transmitem tráfego à VM identificada na linha 5, com base na tabela de fluxos. Essas VMs são responsáveis pela violação das garantias de recepção da VM_C e, por isso, as máquinas físicas que hospedam essas VMs serão notificadas a respeito do congestionamento. Porém, antes de enviar as notificações, o algoritmo 3 identifica quais são as informações a serem enviadas nas mensagens de notificação.

Encontrada a VM_C, o próximo passo é determinar quantos transmissores estão contribuindo com a recepção de tráfego dessa VM. Isso é feito com uma contagem dos fluxos presentes na tabela de fluxos que possuem como valor para o campo de destino o endereço da VM_C. O algoritmo também determina qual a quantidade de banda garantida não utilizada pelas VMs locais que dividem a mesma máquina física com a VM_C (linha 7). Essa quantidade, armazenada na variável U, é calculada com base em uma escolha binária, tendo como resultado a soma das garantias das VMs locais que não possuem qualquer tráfego de recepção no momento da ocorrência do congestionamento. De posse dessas duas informações, o Controlador de Congestionamento envia-as aos transmissores envolvidos no congestionamento. Essas informações são utilizadas pelos

transmissores para determinar qual é a porção do enlace congestionado que eles podem utilizar de forma que o congestionamento seja finalizado.

Quando uma mensagem de notificação é recebida, o controlador de congestionamento executa o algoritmo 4 para definir os limites de envio de tráfego de cada VM local envolvida no congestionamento. Esses limites são definidos de acordo com as informações calculadas no algoritmo 3, que são enviadas em cada mensagem de notificação. O limite de transmissão de cada VM é composto por duas parcelas. A primeira delas está relacionada à garantia de recepção de tráfego da VM_C e a segunda às garantias de recepção não utilizadas pelas demais VMs que compartilham com a VM_C a banda de recepção disponível no enlace congestionado. Ambas as parcelas são calculadas e divididas igualmente entre cada VM que transmite dados à VM_C, de acordo com a descrição a seguir:

- **Parcela referente à garantia de recepção da VM_C:** Quando ocorre um congestionamento, as VMs que possuem demanda referente ao envio de tráfego à VM_C devem compartilhar a garantia de banda contratada pelo proprietário dessa VM. O valor $M.N$, contido nas mensagens de notificação, representa a quantidade de VMs que possuem fluxos de tráfego com destino à VM_C e é utilizado para realizar essa divisão. Caso $M.N$ seja igual a 1, então a garantia de banda de VM_C é atribuída integralmente à única VM que está transmitindo tráfego à VM_C. Em situações onde $M.N$ é maior que 1, a política adotada pelo Gatekeeper-ng é dividir a garantida de banda de recepção da VM_C igualmente entre todas as VMs que enviam tráfego a ela (linha 4). Caso o valor de U, calculado no algoritmo 3, seja igual a 0, apenas essa parcela da banda é atribuída ao limitador de banda RL_R. Dessa forma a vazão agregada do tráfego recebido pela VM_C não ultrapassa a garantia adquirida por seu respectivo cliente e não irá infringir e/ou prejudicar as garantias de tráfego de outros clientes (o que prejudicaria o isolamento do consumo de recursos de rede).

- **Parcela referente à banda do enlace de acesso que não está sendo utilizada por outras VMs:** Para prover um controle de tráfego com conservação de trabalho, o Gatekeeper-ng divide a banda não utilizada da recepção de tráfego do enlace de acesso entre as VMs que possuem demandas além de suas garantias. O Algoritmo 3 extrai da Tabela de Fluxos a quantidade de banda ociosa para a recepção de dados e envia essa informação às VMs que desejam consumir esses recursos. Esse valor é enviado em cada mensagem de notificação de congestionamento, armazenado na variável $M.U$. Para que a divisão dos recursos indicados

Algoritmo 4: Cálculo do Ajuste de Banda

Entrada: Mensagem de Notificação de Congestionamento M, Tabela de Fluxos TF, Capacidade máxima do enlace E, Headroom H

início
  para *cada* VM_R *que, de acordo com a* TF*, envia tráfego para* VM_C **faça**
    $RL_R \leftarrow$ Limitador de Envio associado à VM_R;
    $ParcelaGarantia \leftarrow \left(\frac{VM_R.Garantia}{M.N}\right)$;
    /* Armazenando em $RX\_Controlável$ a largura de banda que pode ser controlada, isto é, que não inclui o Headroom */
    $RX\_Controlável \leftarrow E - H$;
    /* Armazenando em F a proporção entre a banda que está sendo utilizada e a garantia de VM_R */
    $F \leftarrow \frac{VM_R.Garantia}{RX\_Controlável - M.U}$;
    /* A variável $ParcelaLivre$ armazena a quantidade de banda não utilizada que pode ser distribuída entre as VMs que enviam tráfego à VM_C */
    $ParcelaLivre \leftarrow + \left(\frac{F \times M.U}{M.N}\right)$;
    $Limite \leftarrow ParcelaGarantia + ParcelaLivre$;
    /* Após esse passo a divisão de banda do Gatekeeper-ng está completa e a variável $Limite$ armazena a banda que deve ser atribuída ao RL_R */
    $RL_R.Limite \leftarrow Limite$;
    /* Após ajustar o limite, RL_R deve retomar a banda linearmente. */
  fim
fim

em $M.U$ obedeça a proporção das garantias de tráfego cada cliente, o Gatekeeper-ng leva em conta as garantias de recepção de todas as VMs que compartilham o enlace de acesso com a VM_C ao realizar essa divisão. O papel da variável F, presente no algoritmo 4 é armazenar qual é a proporção da garantia de tráfego da VM_C com relação às demais VMs que compartilham a mesma máquina física e que também estão recebendo algum tráfego, utilizando sua garantia de banda

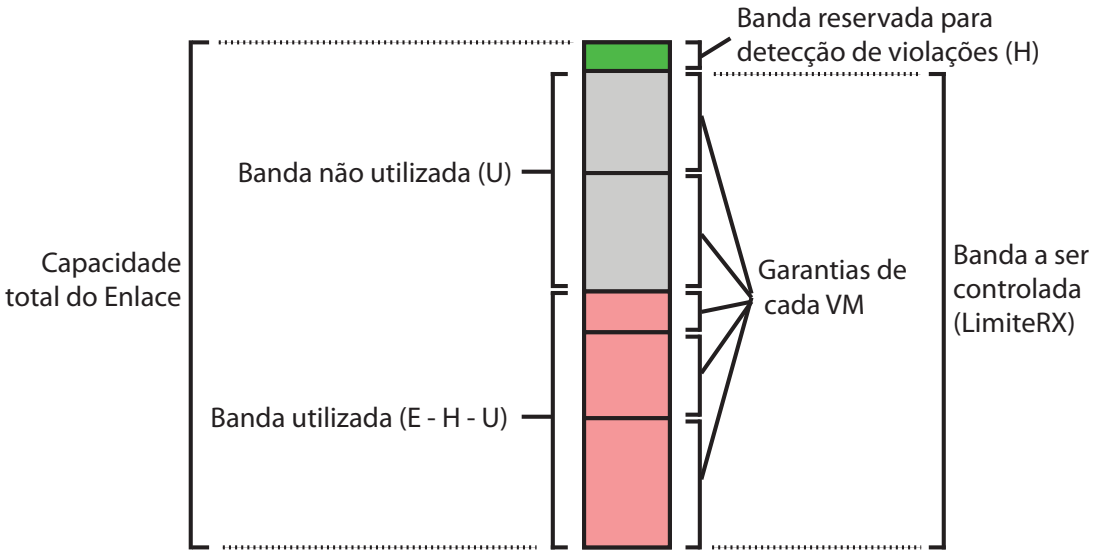


Figura 3.2. Representação gráfica das variáveis utilizadas nos algoritmos 3.

de recepção. Na implementação realizada e também no algoritmo 4, assume-se que as garantias de banda de todas as VMs de um mesmo cliente são iguais e por esse motivo, no algoritmo a $VM_R.Garantia$ é igual à $VM_C.Garantia$[1]. A parcela a ser distribuída entre cada cliente distinto é calculada utilizando-se a proporção armazenada na variável Fe é posteriormente dividida entre todas as VMs (no total de $M.U$) que possuem demandas por envio de tráfego à VM_C (linha 7 do algoritmo 4).

Após o ajuste do limite de envio, cada uma das filas do escalonador de saída volta a incrementar sua taxa de transmissão uma função linear, da mesma forma que a versão original do sistema. Outras funções para retomada de banda podem vir a ser experimentadas, como as utilizadas pelo CUBIC [24] e QCN [31]. No entanto, para os experimentos realizados no capítulo 4, apenas a retomada de banda linear foi utilizada.

3.3 Implementação

Para realizar uma avaliação comparativa entre as duas versões do Gatekeeper, foi implementado um protótipo do sistema proposto nesse trabalho. A mesma plataforma base adotada pela versão anterior do sistema foi reutilizada para a implementação do

[1]Caso essa premissa não seja válida, o algoritmo pode ser alterado para que as mensagens de notificação também incluam essa informação.

Gatekeeper-ng: um ambiente virtualizado baseado em Linux que utiliza o Xen como Monitor de Máquinas Virtuais.

Uma diferença importante entre os ambientes utilizados pelos dois sistemas está na adoção do *Open vSwitch* como *switch* virtual no Gatekeeper-ng, dada a semelhança entre os requisitos da tabela de fluxos da nova solução e a tabela de fluxos do modelo *OpenFlow*. A implementação do sistema foi concluída através da extensão do código de alguns dos componentes do Open vSwitch.

Uma das diferenças entre as duas versões do sistema é a característica mais intrusiva adotada pelo Gatekeeper-ng em relação ao *kernel* do Linux. Ao invés de agregar todas as suas funcionalidades no agente de congestionamento, no nível de usuário, como o Gatekeeper, alguns dos componentes dessa nova versão do sistema são codificados como módulos no nível do sistema operacional, mais próximo dos recursos de hardware das máquinas físicas.

O desenvolvimento de código que executa junto ao Kernel Linux deve ser tratado como código do próprio sistema operacional, obedecendo todas as particularidades do sistema. Isso faz com que o desenvolvedor tenha que observar diferenças importantes, como [36] inexistência de uma biblioteca padrão (*libc*), facilidades para gerência de memória, falta de operações de ponto flutuante, necessidade de considerar problemas de sincronização e condições de corrida, entre outras. Mais detalhes sobre esses aspectos são discutidos no apêndice A. O aprendizado dessas particularidades do sistema operacional Linux foi uma das tarefas mais importantes realizadas durante esse mestrado.

A figura 3.3 apresenta uma visão geral da implementação do Gatekeeper-ng no ambiente virtualizado adotado neste trabalho. A apresentação da implementação é feita de acordo com o ciclo de funcionamento do sistema, que tem início com a monitoração da recepção dos pacotes realizada pelo monitor de tráfego e pela tabela de fluxos. Ligado a essa atividade está a detecção de um congestionamento e a consequente notificação ao controlador de congestionamento sobre o ocorrido. Logo após ocorre a execução dos algoritmos de alocação de banda e o envio das mensagens de notificação aos transmissores envolvidos no problema. O ajuste dos limites dos limitadores de banda dos transmissores e a implementação da retomada da banda "perdida" pelo controlador de transmissão finaliza o ciclo de funcionamento do sistema. A implementação de cada uma dessas funcionalidades é apresentada, na ordem descrita anteriormente, a seguir.

Um elemento importante da implementação é a integração dos mecanismos de cálculo de vazão ao relógio do sistema. A noção da quantidade de tempo percorrido a partir de um determinado instante é importante para muitas funcionalidades do

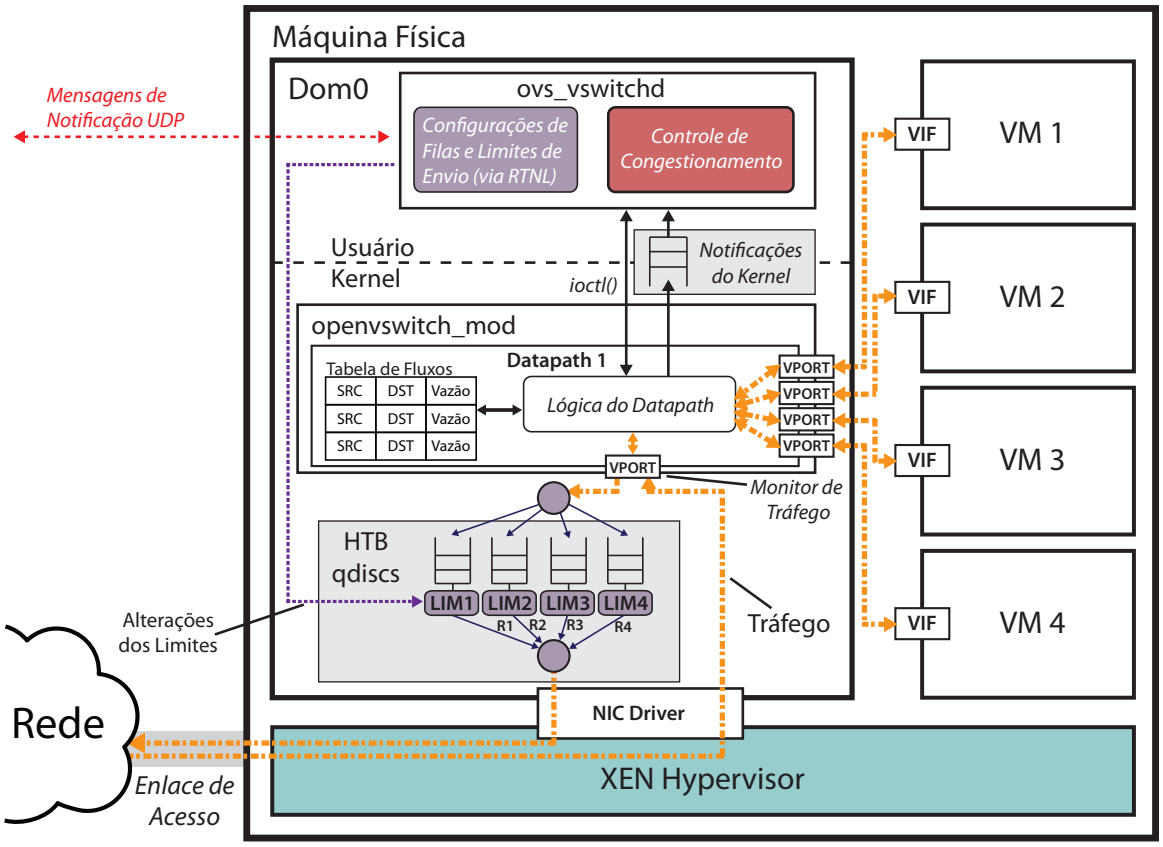


Figura 3.3. Implementação do Gatekeeper-ng. Setas tracejadas que iniciam e terminam no enlace de acesso representam o caminho percorrido pelo tráfego de rede. As setas que cruzam os níveis de Usuário e Kernel representam trocas de dados de controle.

sistema operacional e é essencial tanto para o correto funcionamento de tarefas críticas do sistema, como o escalonamento de processos, quanto para tarefas simples, como por exemplo para identificar a diferença entre um clique duplo do mouse ou dois cliques subsequentes. Entre as tarefas críticas que dependem da informação de tempo oferecida pelo SO também está o controle de tráfego, que tem como unidade fundamental a quantidade de banda consumida (bytes) em um determinado intervalo de tempo (que normalmente é medido em segundos). A discussão a seguir inclui algumas informações relativas ao sistema de controle de tempo do *kernel* do Linux. Uma descrição detalhada dos princípios relacionados se encontra no apêndice B.

3.3.1 Monitor de Tráfego e Tabela de Fluxos

O Monitor de Tráfego do Gatekeeper-ng faz uso direto das variáveis de *kernel* `jiffies` e `HZ` para observar a passagem de tempo e realizar a medição da largura de banda

que flui por cada VPORT. O código que implementa a funcionalidade do monitor de tráfego é apresentado na listagem de código 3.1. A implementação do Monitor de Tráfego é relativamente simples, sendo necessários apenas alguns contadores de bytes (`dp_port.bw`), variáveis para controle de tempo (`dp_port.time`) e uma divisão que é executada periodicamente para calcular a largura de banda. A estrutura `dp_port` representa uma VPORT do *datapath* do Open vSwitch que é "conectada" à uma *vif* no Dom0 do Xen, conforme discutido na seção 2.1.2.

A cada pacote recebido, o código apresentado na listagem de código 3.1 é executado, atualizando o contador de bytes e notificando o processo a nível de usuário caso um congestionamento tenha sido detectado. Ao invés de manter apenas uma medição em cada Monitor de Tráfego, a implementação do Gatekeeper-ng mantém quatro, utilizando a média das últimas medições (*Moving Average*) como o fator determinante para a ocorrência de um congestionamento. Isso é necessário pois o envio de pacotes do Linux pode ser feito em rajadas. Isso é, a cada tique do relógio do sistema o Escalonador de Saída verifica novamente cada fila que possui pacotes a serem enviados. Caso uma das filas tenham pacotes acumulados, vários deles serão enviados de uma só vez até que os tokens no balde dessa respectiva fila sejam todos consumidos, fazendo com que o Escalonador de Saída escolha uma nova fila que tenha tokens disponíveis. Como esse envio em rajadas pode fazer com que o Monitor de Tráfego detecte um congestionamento erroneamente, é utilizada uma média da vazão observada dos últimos 4 intervalos de tempo na tentativa de minimizar esse problema.

```
1  struct dp_port {
2    struct datapath* dp;
3    u32 bw[4];
4    u64 time[4];
5    u8  head;
6    ...
7  }
8
9  void dp_process_received_packet(struct dp_port *p,
10                                  struct sk_buff *skb)
11 {
12   p->bw[p->head]   += skb->len;
13   p->time[p->head]  = jiffies;
14
15   elapsed_time = p->time[p->head] - p->time[prev(p->head)];
16
17   if (elapsed_time >= HZ/100) {
18     moving_average = (p->bw[0] + p->bw[1] + p->bw[2] + p->bw[3])/4;
```

```
19      if (moving_average > RX_THRESHOLD) {
20        /* Congestion */
21        notify_userspace();
22      }
23      p->head = next(p->head);
24      p->bw[p->head] = 0;
25    }
26  }
27
28  struct ovs_skb_cb {
29    struct dp_port    *dp_port;
30    bool               notification_skb;
31  };
32
33  void notify_userspace()
34  {
35    struct sk_buff *skb = alloc_skb(...);
36    ...
37    ((struct ovs_skb_cb)skb->cb)->notification_skb = true;
38    queue_control_packets(skb, queue, ...);
39  }
40
41  flow_used(...);
42  ...
```

Listing 3.1. Implementação dos Monitores de Tráfego, adicionados à cada VPORT do Open vSwitch e implementados junto à Lógica do Datapath, representada na figura 3.3

Uma vez detectado um congestionamento, o Monitor de Tráfego prepara um pacote especial para ser enviado ao nível de usuário, onde o congestionamento será tratado e solucionado. Esse pacote é criado utilizando a estrutura `sk_buff` do Linux, sendo o vetor de armazenamento temporário `cb` utilizado para indicar que esse é um pacote referente à uma notificação de congestionamento. Após criado, o pacote é posto em uma fila (utilizando a função `queue_control_packets`) cujo conteúdo será lido posteriormente pelo Controlador de Congestionamentos.

Após atualizar os dados do Monitor de Tráfego, a função `dp_process_received_packet()` inicia a atualização dos dados da Tabela de Fluxos com a chamada da função `flow_used()`. A implementação da Tabela de Fluxos do Gatekeeper-ng foi feita através da extensão da Tabela de Fluxos *OpenFlow*, que já é utilizada pelo *Open vSwitch*. Campos adicionais foram inseridos em cada entrada da tabela para monitorar a vazão de cada fluxo, conforme descrito na

seção 3.2. Muitas das funcionalidades necessárias para a implementação da Tabela de Fluxos, como a estrutura de dados da tabela de espalhamento e o controle de acesso concorrente à essa estrutura já foram implementados pelo Open vSwitch, tornando a extensão desse componente para monitorar o tráfego uma tarefa simples. A listagem de código 3.2 apresenta de forma simplificada as poucas linhas de código utilizadas para implementar essa funcionalidade.

A estrutura que representa as entradas da Tabela de Fluxos é a `sw_flow`, contendo os dados dos cabeçalhos dos pacotes pertencentes ao fluxo que quando combinados servem para distinguir diferentes fluxos. Além desses, campos que armazenam as estatísticas de cada fluxo, como número de pacotes e o instante em que o fluxo se tornou inativo são mantidos nessa estrutura. Para monitorar a vazão de cada fluxo, foram adicionados os mesmos campos utilizados na implementação dos Monitores de Tráfego (`sw_flow.bw` e `sw_flow.time`).

```
1  struct sw_flow {
2     ...
3     u32 bw[4];
4     u64 time[4];
5     u8  head;
6  }
7
8  void flow_used(struct sw_flow *f, struct sk_buff *skb)
9  {
10    f->bw[f->head]   += skb->len;
11    f->time[f->head]  = jiffies;
12
13    elapsed_time = f->time[f->head] - f->time[prev(f->head)];
14
15    if (elapsed_time >= HZ/100) {
16       f->bw[f->head] = (f->bw[f->head])/elapsed_time;
17       f->head = next(f->head);
18       f->bw[f->head] = 0;
19    }
20 }
```

Listing 3.2. Extensão da Tabela de Fluxos OpenFlow

3.3.2 Algoritmos de Controle de Congestionamento no daemon do Open vSwitch

Os códigos apresentados até aqui são executados no nível do sistema operacional. Na implementação do Gatekeeper-ng buscou-se simplificar ao máximo o trabalho realizado pelo *Kernel* por questões de desempenho e portabilidade, seguindo a mesma prática adotada pelo *Open vSwitch*. O *Kernel* é responsável apenas por manter a monitoração da vazão de cada interface e de cada fluxo e enviar notificações de congestionamentos para o Controlador de Congestionamentos.

O tratamento das notificações de congestionamentos enviadas pelo *Kernel* é implementada à nível de usuário, no daemon do *Open vSwitch*, o `ovs_vswitchd`. Para que os *datapaths*, executados no nível do *Kernel*, possam se comunicar com o daemon, o *Open vSwitch* utiliza filas de notificações que são compartilhadas entre esses dois componentes do sistema, como apresentado na figura 3.3.

Ao invés de verificar continuamente os dados presentes no *Kernel* utilizando chamadas de sistema, como era feito a cada D milissegundos na implementação anterior do Gatekeeper, o *daemon* do *Open vSwitch* registra tratadores de eventos junto ao seu componente que executa à nível de kernel, o `openvswitch_mod`, e passa a dormir, interrompendo a sua execução. Quando um desses eventos ocorre, o `openvswitch_mod` utiliza a fila de notificações para armazenar eventuais informações relacionadas ao evento e acorda o daemon. Após isso, as mensagens na fila de notificações são lidas pelo daemon, que realiza o tratamento desses eventos. Um exemplo de uso dessa interface na implementação original do *Open vSwitch* é a falta de uma regra de chaveamento na sua Tabela de Fluxos quando um pacote de um novo fluxo é recebido. Nesse caso o *datapath* precisa perguntar a um controlador OpenFlow o que fazer com esse fluxo, como bloquear, encaminhar para uma determinada porta ou então enfileirar os pacotes em uma fila específica. A comunicação entre o *datapath* e o controlador é intermediada pelo daemon `ovs_vswitchd`, que utiliza a interface de filas entre os níveis de Kernel e usuário para receber informações relativas ao novo fluxo. O Gatekeeper-ng utiliza essa mesma interface de comunicação para receber notificações do Kernel toda vez que ocorre um Monitor de Tráfego detecta um congestionamento.

Como *daemon* do *Open vSwitch* só é executado quando um evento ocorre, essa regra também se estende ao Controlador de Congestionamento do Gatekeeper-ng, que é implementado dentro desse processo. Essa é uma outra diferença entre essa versão do sistema e da versão anterior, cujo código a nível de usuário era executado a cada D milissegundos mesmo quando não existem fluxos de tráfego no switch virtual.

A implementação do *daemon* do *Open vSwitch* contempla quase todas as fun-

ções do *switch* virtual, como a interface de configuração das interfaces virtuais e dos *datapaths*, manutenção do banco de dados que armazena as informações referentes às configurações dos elementos de chaveamento virtual e o suporte aos canais de comunicação seguros entre controladores *OpenFlow* e *datapaths*. Todas essas funcionalidades são invocadas a partir do laço principal do *daemon*, apresentado na listagem de código 3.3. Basicamente, o daemon realiza as configurações iniciais dos *datapaths* e inicia um laço infinito, que faz com que ele passe a dormir esperando por um evento e acorde para realizar eventuais tratamentos.

```
1  main () {
2     ...
3     while(!exiting) {
4
5        bridge_run();
6        dp_run();
7        netdev_run();
8
9        signal_wait(sighup);
10       bridge_wait();
11       dp_wait();
12       netdev_wait();
13       poll_block();
14    }
15    ...
16 }
```

Listing 3.3. Visão geral da implementação do daemon do Open vSwitch, ovs_vswitchd

De todos os fluxos de execução que têm início no laço infinito do *daemon*, a chamada de função importante para o Gatekeeper-ng é a `bridge_run()`. Essa função é o ponto de partida para a lógica básica de um *switch* que implementa a especificação *OpenFlow*, chamado pelo *Open vSwitch* de `ofproto`. A estrutura que armazena as variáveis referentes ao *switch* virtual, como endereço do controlador *OpenFlow* associado ao *datapath* e demais variáveis de configuração do *switch,* é a `struct ofproto`. Essa estrutura também foi alterada para abrigar as variáveis de controle do Gatekeeper-ng: o limite que indica qual é banda reservada para detecção de congestionamentos, ou *headroom* (identificada pelo limiar `rx_threshold` nas listagens de código); as estruturas referentes aos *sockets* para comunicação entre Controladores de Congestionamentos instalados em outras máquinas físicas; e a estrutura do serviço de diretórios, que contém as informações referentes às VMs.

Na função `ofproto_run()`, apresentada na listagem de código 3.4, referente à execução do `ofproto`, foram inseridos os códigos relacionados ao Controlador de Congestionamentos do Gatekeeper-ng para o tratamento das mensagens de notificação advindas do *Kernel*, envio e recebimento de notificações de congestionamento de/para outros Controladores de Congestionamento e ajuste de taxas do Escalonador de Saída. O tratamento das notificações do *Kernel* é feito pela função `handle_kernel_msg()`, que verifica qual foi o tipo do evento enviado pelo *Kernel* e caso esse seja uma notificação de congestionamento, realiza a leitura da Tabela de Fluxos utilizando a chamada de sistema `ioctl()` e executa a função `gk_feedback()` que irá determinar o destino da notificação de congestionamento. Nessa mesma listagem de código também estão as chamadas das funções para receber uma mensagem de congestionamento de um outro Controlador de Congestionamento, através chamada de sistema `recvfrom()` e no processamento dessa mensagem recebida, realizado pela função `gk_recv_msg()`. A comunicação entre os Controladores de Congestionamento é feita através de mensagens UDP.

```
1  /* From bridge_run() */
2  ofproto_run(struct ofproto *p) {
3    ...
4
5    ret = recvfrom(p->rsock, buf, ...);
6    if (ret > 0) {
7      gk_recv_msg(p, buf);
8    }
9
10   poll_fd_wait(p->rsock, POLLIN);
11
12   for (;;) {
13     ret = dpif_recv(p->dpif, buf);
14     if (ret)
15       handle_kernel_msg(p, buf);
16     else
17     break;
18   }
19   ...
20 }
21
22 handle_kernel_msg(struct ofproto *p, struct ofpbuf *msg) {
23   struct odp_flow *flows
24   switch (msg->reason) {
25     ...
```

```
26        case GK_CONGESTION:
27          n_flows = gk_get_flows(p, &flows);
28          gk_feedback(p, &flows, n_flows);
29          break;
30      }
31      ...
32  }
```

Listing 3.4. Fluxos para chamadas de funções do Gatekeeper-ng, implementadas no ovs_vswitchd

3.3.3 Envio e recebimento de notificações

Quando ocorre um congestionamento na recepção do enlace de acesso de uma máquina física, a tarefa do Gatekeeper-ng é simplesmente prover informações referentes ao congestionamento às VMs transmissoras que estão causando o problema. O envio de notificações de congestionamento é feito pela função `gk_feedback()`, que decide para quem a mensagem de notificação deve ser enviada. Conforme discutido na apresentação do algoritmo 3, o Gatekeeper-ng envia notificações para cada uma das VMs que estão enviando tráfego à VM local que está excedendo a sua garantia em maior quantidade. Isso é feito com o auxilio da estrutura do serviço de diretórios `hdir` armazenada na estrutura `ofproto`. Em um primeiro momento a função `update_hdir()` atualiza as informações sobre cada uma das VMs locais mapeadas em `hdir`. A função `most_unfair_rx` então busca a VM local que está excedendo sua garantia de tráfego em maior quantidade. A quantidade de banda disponível e o número de VMs contribuindo para o congestionamento são armazenados nas variáveis `idle_bw` e `n_senders` respectivamente. Esses valores são encapsulados na mensagem de notificação representada pela variável `fb_msg` e uma cópia dessa é enviada às VMs que enviam tráfego à VM retornada pela função `most_unfair_rx(hdir)`.

```
1  gk_feedback(struct ofproto *p,
2              struct odp_flow **flows,
3              size_t n_flows) {
4    ...
5    struct gk_hostdir *hdir = p->hdir;
6    struct gk_host    *h;
7    struct gk_fb      *fb_msg;
8
9    update_hostdir(flows, n_flows, hdir);
10   idle_bw = get_idle_bw(hdir);
```

```
11      ...
12      /* Looking for the vm which is exceeding
13         the most its RX guarantee */
14      h          = most_unfair_rx(hdir);
15      n_senders = senders(hdir, h);
16      ...
17      fb_msg->h          = h;
18      fb_msg->n_senders = n_senders;
19      fb_msg->idle_bw    = idle_bw;
20      ...
21      foreach(/* vm V in flows sending traffic to h */) {
22        gk_send_feedback(V, fb_msg);
23      }
24  }
```

Listing 3.5. Implementação do envio de notificações do Gatekeeper-ng, codificada junto ao daemon do Open vSwitch, o ovs_vswitchd, representado na figura 3.3

Atualmente o Gatekeeper envia uma mensagem de notificação *unicast* para cada uma das máquinas físicas que hospedam as máquinas virtuais envolvidas em um congestionamento. Uma ideia para reduzir o envio de mensagens duplicadas e consequentemente reduzir o processamento dessa tarefa seria configurar diversos endereços *multicast* no datacenter, um para cada cliente. Dessa forma a máquina física que envia o a notificação precisaria enviar apenas uma mensagem de notificação para o endereço *multi-unicast* do cliente envolvido no congestionamento. A verificação de quais VMs devem ter sua transmissão limitada pelo escalonador de saída seria distribuída entre cada máquina física que faz parte desse endereço *multicast*.

O procedimento executado no recebimento de uma notificação de congestionamento é apresentado na listagem de código 3.6. A função `gk_recv_msg()` implementa a divisão da banda garantida do cliente acrescida de uma eventual parcela da banda livre no enlace de acesso da VM receptora, conforme descrito na apresentação do algoritmo 4. Após calcular o valor referente ao limite de tráfego a ser imposto para a VM transmissora, o Controlador de Congestionamentos envia essa informação para o Escalonador de Saída do Gatekeeper-ng.

No protótipo anterior do Gatekeeper a recuperação de informações do Escalonador de Entrada e a alteração dos limites das filas do Escalonador de Saída eram feitos utilizando um programa auxiliar — a ferramenta do Linux chamada `tc`. Esse programa fazia a leitura dos dados das filas de entrada e entregava-os ao Agente de Congestionamento em forma de texto, que era então processado para a extração dos

dados relevantes. Esse procedimento era muito caro computacionalmente, pois o programa Python que implementava o Agente de Congestionamento fazia pelo menos duas chamadas de sistema `fork()` para criação de novos processos responsáveis pela leitura e alteração das informações dos Escalonadores.

Na implementação realizada, o Controlador de Congestionamento utiliza a mesma interface de comunicação com o Kernel que é utilizada pelo `tc`, chamada de RTNETLINK. Essa interface de comunicação é fornecida pelo Linux para que processos a nível de usuário possam alterar parâmetros dos subsistemas de redes do sistema operacional, como por exemplo rotas e os endereços das interfaces de rede. A NETLINK [49] é uma extensão da implementação padrão da interface de sockets do Linux e, portanto, a comunicação entre o processo a nível de usuário e o Kernel deve ser feito através de trocas mensagens. A função `gk_rtnl_change_rate()` apresentada na listagem de código 3.6 é responsável por iniciar a conexão com o Kernel, montar a mensagem contendo os parâmetros para a alteração dos limites das filas e enviar a requisição utilizando o *socket* aberto, através da função `rtnl_send()`.

```
1  gk_recv_msg(struct ofproto *p, struct gk_fb *fb_msg) {
2    struct gk_hostdir *hdir;
3    ...
4    foreach(/* vm V in hdir sending traffic to fb_msg->h */) {
5      target_rate = V.guarantee + fb_msg->idle_bw *
6          V.guarantee / (p->rx_threshold - fb_msg->idle_bw);
7      target_rate = target_rate / fb_msg->n_senders;
8      gk_rtnl_change_rate(p, h, target_rate);
9    }
10 }
11 ...
12 gk_rtnl_change_rate(struct ofproto *p,
13                     struct gk_host *h,
14                     uint32_t rate) {
15    struct rtnlreq req;
16    struct tc_htb_opt opt;
17    ...
18    /* Set req type and attributes according to
19       the RTNL spec for HTB msgs */
20    rtnl_send(p->rtnlsock, &req);
21 }
```

Listing 3.6. Implementação do recebimento de notificações do Gatekeeper-ng, codificada junto ao daemon do Open vSwitch, o ovs_vswitchd, representado na figura 3.3

3.3.4 Mecanismo de Incremento da Taxa de Transmissão

Assim como na versão original do Gatekeeper, o Escalonador de Saída é implementado utilizando a *Queuing Discipline* HTB. O diferencial na implementação do Gatekeeper-ng esta na lógica da retomada de banda, que foi implementada diretamente no código do HTB. Isso evita a necessidade que um processo a nível de usuário fique ajustando o limite de cada fila a cada D milissegundos, como era feito na implementação anterior do sistema. Essas alterações no módulo do HTB são apresentadas na listagem de código 3.7.

Antes de entrar em detalhes a respeito do código de retomada de banda, é pertinente introduzir alguns dos detalhes de implementação do HTB. Conforme discutido na seção 2.2, essa *Queuing Discipline* do Linux implementa um *token bucket* hierárquico. Na implementação do HTB cada classe tem dois um baldes de *tokens*, sendo um para o limite mínimo garantido para a classe e um para seu limite superior. O consumo de *tokens* desses baldes é feito de acordo com o tamanho de cada pacote, isto é, quanto maior o pacote, maior o número de tokens consumidos de cada balde. Para implementar esse consumo de tokens de maneira eficiente, o HTB utiliza frações de tempo para representar os tokens e mantém tabelas de consulta rápida (*lookup tables*) que relacionam quantidade de tokens que devem ser retirados do balde com o limite de tráfego especificado para classe e o tamanho de um pacote. Isto é, dado que um pacote de X bytes será transmitido em uma fila que possui um limite de tráfego de Y bytes por segundo, então o consumo de tokens em termos do módulo HTB é dado pela equação 3.1

$$Tokens = \frac{X}{Y} \times \frac{1}{Tokens\ em\ 1\ Segundo} \tag{3.1}$$

A tabela de consulta rápida é utilizada por dois motivos principais. O primeiro deles é o desempenho obtido com a utilização de valores pré-calculados ao invés de recalcular o consumo de *tokens* para cada pacote. Essa otimização é válida no sentido de que o limite configurado para cada fila não era alterado frequentemente. O segundo motivo é a inexistência do suporte à valores de ponto flutuante no Kernel Linux, dificultando a manipulação de valores fracionados como os valores envolvidos nas frações da equação 3.1. Por esse motivo essa tabela é calculada no momento da criação da *qdisc* pela própria ferramenta `tc`, que é executada no nível de usuário e pode utilizar aritmética de ponto flutuante.

Como o limite de cada fila é alterado frequentemente para a implementação da retomada de banda do Gatekeeper-ng, foi necessário encontrar um método alternativo para calcular o consumo de *tokens* e evitar a reconstrução da tabela de consulta rápida

a cada incremento no limite da fila. Esse era o procedimento realizado pela versão original do Gatekeeper através o uso da ferramenta `tc`, que era executada para cada fila a cada 10 milissegundos. Na implementação do Gatekeeper-ng, quando limite é alterado, o HTB calcula o consumo de tokens diretamente, deixando de lado a tabela de consulta rápida. É feita uma manipulação dos valores envolvidos na operação do cálculo de consumo de tokens para evitar *overflows* e *underflows* de números inteiros devido à falta de suporte a valores de ponto flutuante no Kernel Linux. Essa manipulação é apresentada de forma simplificada na listagem de código 3.7. Basicamente, o calculo do consumo de tokens é feito utilizando um fator de multiplicação, que é dependente da frequência de tiques do relógio do sistema a cada segundo, pois o número de *tokens* adicionados ao balde a cada tique também depende desses dois fatores, que são mantidos pelo subsistema de tempo do Linux. As modificações no HTB feitas para o Gatekeeper-ng envolvem a leitura desses fatores, o cálculo de um coeficiente de multiplicação (representado pela variável `nshift`), e o uso de variáveis de 64 bits. A variável que armazena o consumo de tokens relacionadas a um pacote a ser transmitido é a `pkt2toks`.

Para a implementação da retomada de banda do Gatekeeper-ng foram adicionada duas variáveis `trate` e `crate` à estrutura que representa uma classe HTB. Essas variáveis controlam o limite atual e o limite superior que foram alterados pelo Controlador de Congestionamento. A cada pacote transmitido, o HTB utiliza a função `htb_charge_class()` para atualizar o número de tokens nos baldes de cada classe da hierarquia de classes. As funções `htb_accnt_tokens()` e `htb_accnt_ctokens()` são responsáveis por remover os tokens dos dois baldes que contém os limites mínimo e superior de cada classe, respectivamente. A leitura do consumo de tokens a partir da tabela de consulta rápida é feita com a função `qdisc_l2t()`. A função `gk_inc` é utilizada para alterar os valores das variáveis `trate` e `crate` com o passar do tempo, retornando um valor não nulo para informar as funções `htb_accnt_tokens()` e `htb_accnt_ctokens()` se o consumo de *tokens* deve ser calculado novamente ou não.

```
1 struct htb_class {
2   ...
3   struct qdisc_rate_table *rate;  /* rate table of the class itself */
4   struct qdisc_rate_table *ceil;  /* ceiling rate (limits borrows too)
        */
5   long buffer, cbuffer;            /* token bucket depth/rate */
6   u32 trate;                      /* Target rate */
7   u32 crate;                      /* Current rate */
8 }
```

```
 9  ...
10  htb_charge_class(... htb_class *cl, struct sk_buff *skb) {
11    ...
12    bytes = skb_len(skb);
13    while (cl) {
14      htb_accnt_tokens(cl, bytes);
15      htb_accnt_ctokens(cl, bytes);
16      cl = cl->parent;
17    }
18  }
19
20  htb_accnt_tokens(struct htb_class *cl, int bytes) {
21    u32 pkt2toks;
22
23    pkt2toks = (long) qdisc_l2t(cl->rate, bytes);
24    if (gk_inc(cl, cl->rate)) {
25      pkt2toks = (long)
26        (((sec2ticks*bytes)/cl->crate)*tick2toks_shifted)>>nshift);
27    }
28    cl->tokens -= pkt2toks;
29  }
30
31  gk_inc(struct htb_class *cl, struct qdisc_rate_table *rtab) {
32    u64 now;
33
34    if (cl->crate >= rtab->rate.rate)
35    return 0;
36
37    now = get_time();
38    if (cl->timer < now) {
39      while (cl->timer < now) {
40        cl->crate += GK_INC;
41
42        /* Time passed, rate increased */
43        if (cl->crate >= rtab->rate.rate)
44          return 0;
45      }
46    }
47    return 1;
48  }
```

Listing 3.7. Implementação da função de retomada de banda no HTB

A forma com que é feita o incremento do limite de banda de cada classe HTB finaliza a descrição da implementação do Gatekeeper-ng. O próximo capítulo conti-

nua com a descrição dos experimentos realizados para comparar as duas versões do Gatekeeper.

Capítulo 4

Avaliação

O capítulo anterior apresentou a nova implementação para o Gatekeeper concluída neste trabalho. Composta por uma nova arquitetura e novos algoritmos de controle de tráfego, essa implementação explora recursos do *Open vSwitch* e outros elementos de programação no *kernel* para garantir um melhor desempenho e uma maior escalabilidade para a solução de controle de tráfego proposta.

Este capitulo apresenta uma avaliação comparativa entre os dois sistemas estudados neste trabalho. O objetivo da experimentação foi apontar as vantagens e desvantagens de cada sistema, considerando seu comportamento em diversos padrões de tráfego. As métricas envolvidas na avaliação foram o custo de processamento e a precisão dos algoritmos de escalonamento de tráfego implementados. As próximas seções detalham os experimentos realizados mostrando que a nova implementação é capaz de atingir maior precisão no controle de tráfego, impondo menor sobrecarga de processamento para as máquinas físicas.

4.1 Considerações Preliminares

O ambiente de testes utilizado para realização dos experimentos possui 5 máquinas com um processador de quatro núcleos e placas de rede de 1 Gbps, todas ligadas a um único *switch*. O número de máquinas virtuais executadas em cada máquina física foi limitado a três para evitar quaisquer interferências nos resultados relacionadas ao escalonador de VMs do Xen [33, 64]. Isto é, para cada uma das VMs em uma máquina física é reservado um dos núcleos de processamento, que é utilizado exclusivamente por essa VM.

Os padrões de tráfego presentes na experimentação foram gerados utilizando o *micro-benchmark* `netperf` e uma variação do mesmo que é capaz de gerar tráfego in-

termitente. Todos os experimentos têm duração de 20 segundos e envolvem a indução de um congestionamento utilizando algum padrão de tráfego gerado pelas VMs. Para cada experimento foram realizadas 30 repetições e os resultados contidos neste capítulo apresentam intervalos de confiança de 99%. A exceção está na comparação do comportamento interno dos dois sistemas (seção 4.2), que apresenta as medições realizadas em apenas uma execução.

Nessa avaliação foram consideradas duas formas de realizar as medições de tráfego para comparar a quantidade de recursos utilizados por cada VM. A primeira delas seria utilizar o valor reportado pelo *micro-benchmark*, que indica qual foi a quantidade de bytes efetivamente transmitidos entre cada aplicação durante a execução do experimento (*goodput*). Porém, como estamos controlando o acesso aos recursos de rede independente do comportamento de qualquer protocolo, a forma de medição adotada para a avaliação foi a quantidade de dados efetivamente inseridos na rede (*throughput*), o que incluí também os cabeçalhos dos pacotes. Dessa forma as medições não contam com eventuais perdas de dados que não são capturadas pela medição de *goodput*. A vazão do tráfego utilizada por cada VM foi monitorada no Dom0 de cada máquina física, através da leitura da quantidade de dados que atravessa cada interface de rede.

Um dos problemas enfrentados durante a realização dos experimentos foi a ausência de um relógio global para determinar o início e o fim de cada transmissão. Como os experimentos também envolvem tráfego que não possui controle de congestionamento, mesmo um pequeno atraso no início da transmissão de um fluxo concorrente TCP, por exemplo, pode tornar a sua vazão nula quando nenhum mecanismo de controle de tráfego é utilizado. Isso acontece porque o controle de congestionamento do TCP é ativado no início da fase *slow start*, impedindo que ele consiga atingir uma boa taxa de transmissão. Para contornar esse problema, os relógios das máquinas foram sincronizados utilizando o protocolo *NTP* [40] e os experimentos, ao invés de serem iniciados remotamente, são agendados para serem iniciados ao mesmo instante utilizando o escalonador de tarefas do Linux, o *cron*. Com isso, no ambiente em questão, as diferenças de tempos de disparo se tornam desprezíveis em relação aos tempos de execução.

A versão original do Gatekeeper possui uma limitação com relação à capacidade máxima de tráfego que ela consegue controlar [54, 55]. Por estar implementada no próprio Kernel do Linux, a nova versão do sistema não possui essa limitação, sendo capaz de processar o tráfego de rede à velocidade máxima do enlace de acesso [48]. No entanto, para que seja possível realizar execuções pareadas [30] na avaliação comparativa entre os dois sistemas, o limite máximo para o controle de banda do Gateekeper-ng foi reduzido para o mesmo que a versão original do sistema, 860 Mbps. Esse valor não contempla a quantidade de banda reservada para detecção de congestionamentos (ou

headroom), que no Gatekeeper-ng foi configurada como 940 Mbps.

4.2 Comportamento Comparado dos Dois Sistemas

O objetivo desse primeiro experimento foi avaliar o comportamento dos dois sistemas de controle de tráfego ao longo do tempo na presença de congestionamentos. Para isso, o ambiente de testes foi configurado com dois clientes, A e B, cujas garantias de banda são 70% e 30% da capacidade do enlace respectivamente. O cliente A possui apenas duas VMs e o cliente B, quatro. O posicionamento das VMs no ambiente de testes e o padrão de tráfego utilizado para induzir o congestionamento são apresentados na figura 4.1.

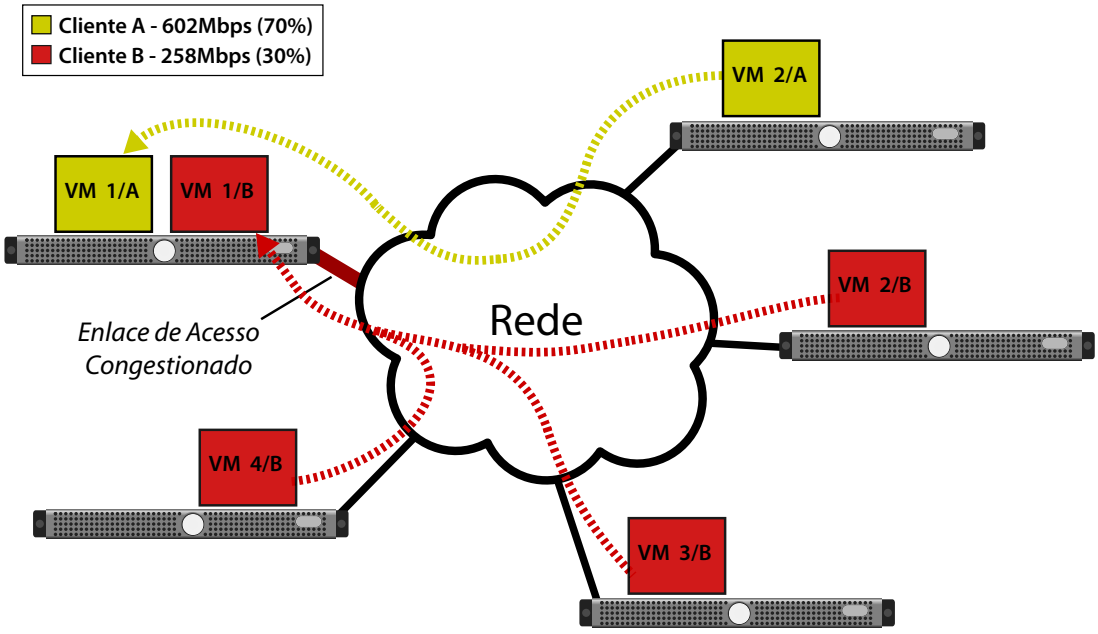


Figura 4.1. Experimento para Avalizar o Comportamento dos dois Sistemas

Nesse experimento o tráfego gerado pelos clientes é composto pela combinação de uma única conexão TCP e vários fluxos UDP. O tráfego gerado entre as VMs do cliente A utiliza o protocolo protocolo TCP e o tráfego trocado entre as VMs do cliente B utiliza o protocolo UDP. O motivo para essa combinação de protocolos é a característica distinta como eles lidam com eventuais congestionamentos. Como UDP não possui controle de congestionamento, ele continuará enviando tráfego, independente de eventuais perdas de pacotes que possam acontecer. Já o TCP, por sua vez, apresenta comportamento responsivo à eventuais perdas de pacotes diminuindo sua taxa de envio para tentar reduzir o congestionamento na rede, como discutido na seção 2.2. Caso o

sistema de controle de tráfego não tenha capacidade de controlar a alocação de banda adequadamente, o consumo de recursos do cliente A estará comprometido. Esse cenário busca reproduzir o caso em que um dos clientes possui uma aplicação bem-comportada (cliente A) que coopera para uma divisão justa dos recursos da rede e um outro cliente egoísta (cliente B), que pretende utilizar os todos recursos disponíveis apenas em benefício próprio, sem se importar com as demandas de outros clientes. Esse experimento é semelhante ao utilizado no artigo que apresenta o Seawall [53], para medir a capacidade que do sistema de evitar ataques de negação de serviço. Em nosso caso, o tráfego gerado pelo cliente B poderia ser caracterizado como um ataque.

Como o objetivo dos dois sistemas é alocar a banda disponível no enlace de rede congestionado de acordo com as garantias, a métrica de interesse nesse experimento foi a quantidade de tráfego efetivamente recebida pelas VMs 1/A e 1/B. Para isso, foi monitorada a vazão do tráfego das interfaces virtuais de cada uma das máquinas virtuais em intervalos de tempo regulares de 10 milissegundos. O resultado da monitoração referentes às VMs 1/A e 1/B é apresentado nos gráficos de utilização de banda da figura 4.2 . À esquerda está a vazão observada quando o Gatekeeper original é utilizado e à direita quando o Gatekeeper-ng é utilizado.

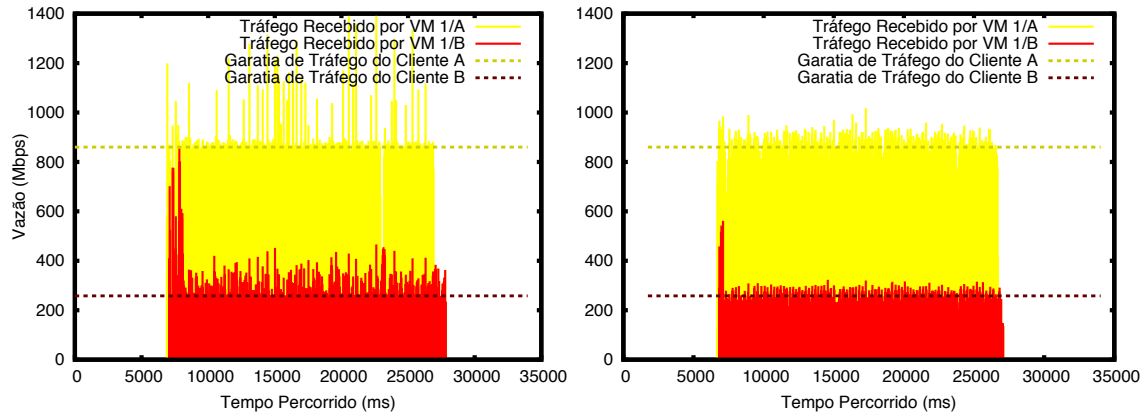
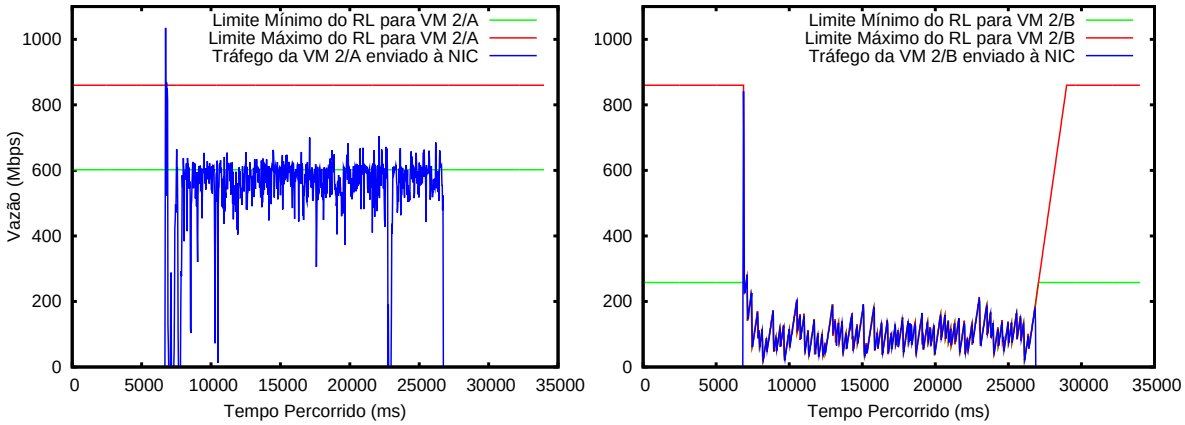


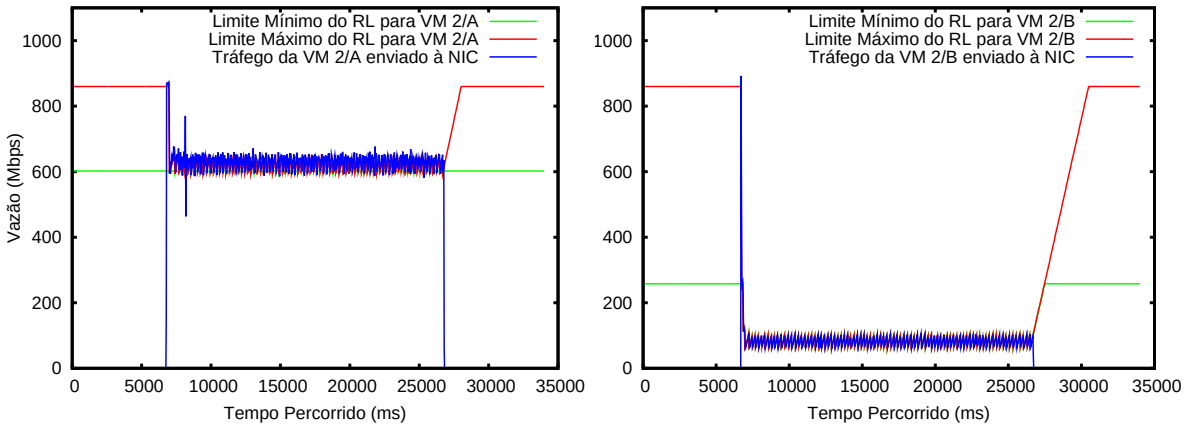
Figura 4.2. Recepção de Tráfego da VM 1/A e VM 1/B utilizando o Gatekeeper (à esquerda) e o Gatekeeper-ng (à direita)

Além da vazão observada na interface de rede de cada máquina virtual, também foram monitorados os valores dos parâmetros de cada classe HTB do Escalonador de Saída associada a cada máquina virtual geradora de tráfego. A configuração desses parâmetros deveria respeitar a divisão de banda ideal considerando o padrão de tráfego gerado, sendo 70% para a VM 2/A e os 30% do cliente B divididos em parcelas de 10% para cada uma das suas VMs. Os resultados capturados por esse monitoramento são apresentados na figura 4.3. As medições das classes HTB relacionadas às VM 3/B e

VM 4/B foram omitidas pois são muito semelhantes aos valores medidos para a VM 2/B.



(a) Tráfego da VM 2/A (esquerda) e VM 2/B (direita) utilizando o **Gatekeeper**



(b) Tráfego da VM 2/A (esquerda) e VM 2/B (direita) utilizando o **Gatekeeper-ng**

Figura 4.3. Resultados dos Experimentos Comparando o Comportamento dos dois Sistemas

Esses resultados mostram que a divisão de tráfego realizada pelo Gatekeeper-ng se mantém mais próxima da divisão ideal, diminuindo as violações das garantias de tráfego que ocorrem quando a versão original do sistema é utilizada. O Gatekeeper original está sujeito a oscilações maiores pela imprecisão inerente ao mecanismo de identificação de fluxos não responsivos pelos descartes observados. Essas oscilações são responsáveis pelo desempenho do Gatekeeper original abaixo dos limites ideais estabelecidos na figura 2.18 quando pelo menos um dos fluxos é TCP. Ao longo do tempo, o controlador no receptor vai identificar pacotes das diversas fontes, que serão notificados do congestionamento e reduzirão sua taxa de transmissão em momentos independentes.

Já no Gatekeeper-ng, o controlador no receptor pode notificar simultaneamente todos os transmissores, que agem de forma mais coordenada.

Além disso, como o Gatekeeper-ng é capaz de determinar com maior precisão o consumo e as demandas de cada um dos clientes, os Controladores de Congestionamento podem ajustar a banda diretamente para o valor ideal de acordo com o tráfego observado. Isso dispensa a necessidade de utilizar um ciclo de controle para ajustar a banda iterativamente, com múltiplas reduções à metade do valor anterior, como é feito pelo Gatekeeper original. Ao receber uma notificação de que há congestionamento e que há três fluxos do cliente B no enlace sobrecarregado, os limites mínimo e máximo são reduzidos a um terço do limite mínimo original, devido ao compartilhamento.

Um último detalhe digno de nota com relação à figura 4.3 se refere à forma como as duas versões de Gatekeeper atuam sobre o fluxo TCP (tráfego da VM 2/A) para mantê-lo dentro dos limites estabelecidos. No Gatekeeper original, os descartes extras causados pelo escalonador de entrada são usados para manter a conexão sob controle, aproveitando o comportamento de TCP. Dessa forma, os limites mínimo e máximo do escalonador de saída no transmissor não chegam a ser alterados em nenhum momento. Já com o Gatekeeper-ng, já que não há descartes extras causados pelo escalonador, o ajuste do transmissor é realizado diretamente pela redução do limite máximo permitido para a conexão. Nesse caso, como há apenas um fluxo envolvido, o limite mínimo não precisa ser reduzido.

4.3 Estabilidade

Um outro experimento importante na comparação dos dois sistemas foi a medição de quão estáveis são os algoritmos de alocação de banda frente a uma grande quantidade de fluxos e/ou clientes concorrentes. Para realizar essa avaliação, esse experimento monitorou o comportamento dos dois sistemas à medida que o número de fluxos participantes e de fluxos no congestionamento induzido eram variados.

Utilizando os recursos disponíveis, o ambiente de testes foi configurado com 3 clientes A, B e C. O número de fluxos que concorrem simultaneamente pelo mesmo enlace de acesso foi variado entre 2 e 7. A geração desses fluxos foi dividida entre as VMs dos três clientes, cujas quantidades e posicionamento são presentadas na figura 4.4.

O incremento dos fluxos e de clientes participantes no congestionamento a cada passo do experimento foi feito da seguinte forma: em todos os experimentos, o cliente A estabelece um fluxo da máquina 2/A para a máquina 1/A. Em um primeiro momento, apenas o cliente B compete com o tráfego do cliente A. Para isolar a contenção apenas

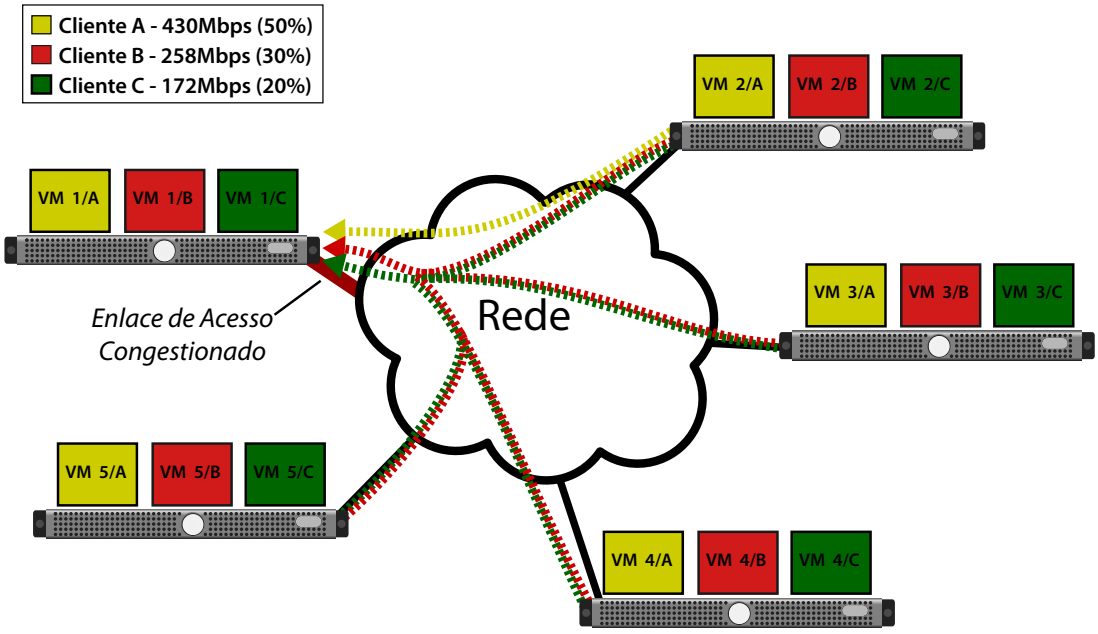


Figura 4.4. Experimento para Avalizar a Estabilidade dos dois Sistemas

ao enlace de acesso à máquina destino, o cliente B gera tráfego a partir das suas 3 VMs que não compartilham a mesma máquina física com a VM que gera o tráfego TCP, a VM 2/A: primeiro, apenas um fluxo, de 2/B para 1/B; depois, dois fluxos, de 2/B e de 3/B, para 1/B e, finalmente, três fluxos, de 2/B, 3/B e 4/B para 1/B. Em seguida, com o cliente B mantendo três fluxos concorrentes, o cliente C também passa a gerar tráfego seguindo os mesmos passos de B em relação ao número de fluxos. Dessa forma, o número de fluxos que concorrem com o fluxo de A pelo enlace de acesso à máquina destino varia de 1 a 6. Na três primeiros casos, quando apenas 2 clientes competem pelo enlace de acesso, o resultado esperado é que cada um deles utilize a sua garantia de banda e uma parte dos recursos não utilizados pelo cliente C. Quando C começa a utilizar a sua parte dos recursos, a partir do caso com quatro fluxos concorrentes, o sistema de alocação de banda deveria ser capaz de manter a utilização do enlace de acordo com as garantias de cada cliente.

Seguindo a mesma linha de raciocínio do experimento anterior, o tráfego gerado por um dos clientes, A, utiliza fluxos TCP, enquanto as VMs dos outros clientes utilizam UDP. Esse padrão de tráfego evidencia os problemas na alocação de banda dos dois sistemas que, caso seja ineficiente, prejudicará o cliente que está utilizando um protocolo com controle de congestionamento. A alocação da banda do enlace de acesso à máquina compartilhada é feita com 430 Mbps para o cliente A, 258 Mbps para B e 172 Mbps para C.

Os fluxos foram mantidos por 30 segundos e a métrica utilizada para medir a qualidade dos algoritmos foi a vazão do tráfego atingida por cada um dos clientes durante todo o experimento. A medição desse valor é feita pelo Dom0 da máquina física que possui o enlace congestionado utilizando um contador de bytes transmitidos presente em cada interface virtual (*vif*). Para obter a vazão, são feitas duas leituras desse contador: uma logo após o início do experimento e outra no seu término. Com a diferença entre os dois valores lidos e o intervalo de tempo entre as leituras é possível obter a vazão do tráfego recebido pelas VMs. Os resultados desse experimento são apresentados na figura 4.5. Como na figura 2.18, as linhas horizontais indicam a vazão alocada para cada cliente.

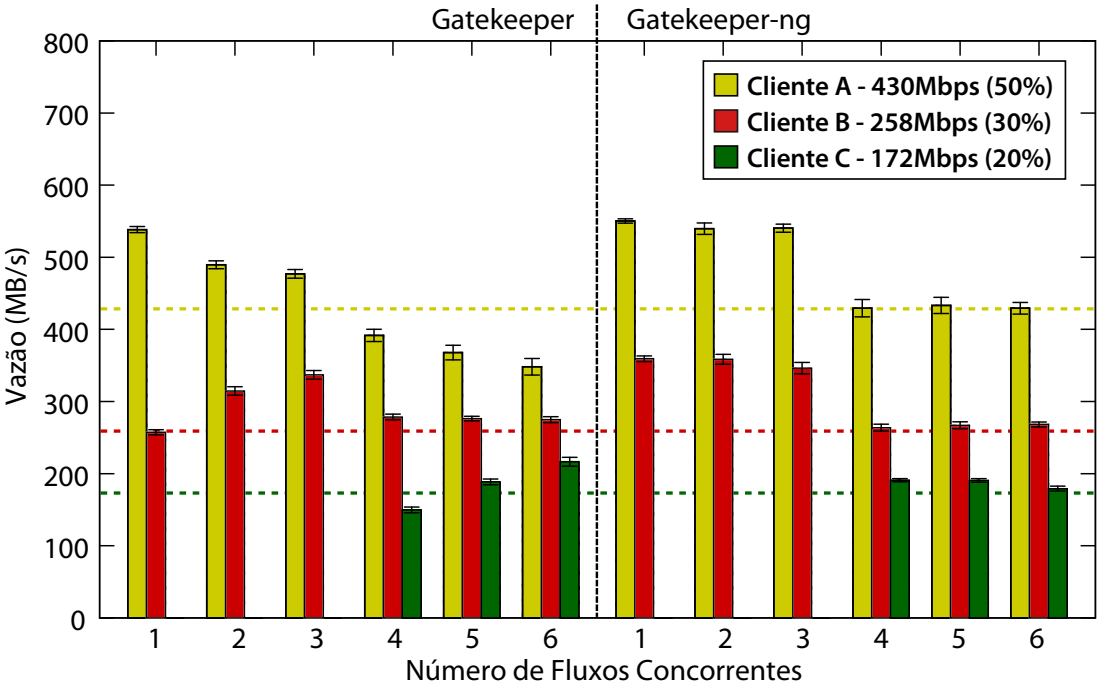


Figura 4.5. Resultados dos Experimentos Comparando a Estabilidade dos dois Sistemas

Claramente, Gatekeeper-ng apresenta um desempenho muito mais estável em todos os casos, apresentando uma distribuição de recursos como desejada em cada caso. Já o sistema original tem um desempenho aceitável enquanto apenas dois clientes competem pelos recursos, apesar do comportamento não ser constante: a divisão dos recursos excedentes é afetada pelo número de fluxos concorrentes, o que não seria o esperado em um mecanismo de reserva de banda por cliente. Seria mais desejável se a divisão da banda entre clientes fosse previsível em função apenas do excesso disponível, e não do comportamento dos clientes. Além disso, o volume total de recursos utilizados

é claramente superior no caso do Gatekeeper-ng para os casos com apenas dois clientes, pois as taxas obtidas são sempre iguais ou superiores às obtidas no sistema original.

Quando um terceiro cliente é acrescentado ao cenário, os mecanismos para distribuição de recursos entre os clientes não são capazes de manter a garantia de banda do cliente A, que usa TCP. Nesse caso, a contenção no núcleo da rede causa perdas extras ao fluxo TCP, que reage reduzindo sua taxa de transmissão. A banda não utilizada por por TCP nesse caso acaba sendo consumida pelo cliente C de forma indevida. Já o comportamento da nova versão é estável, garantindo os limites mínimos previstos e até permitindo alguma utilização extra por parte de C nos casos com quatro e cinco fluxos concorrentes. Isso é possível uma vez que no Gatekeeper-ng o *headroom* definido no receptor não implica no descarte de pacotes que ultrapassem esse limite, apenas causa a atuação do mecanismo de controle após aquele momento.

4.4 Avaliação com padrões de tráfego diversos

Os experimentos anteriores consideraram sempre um padrão de tráfego onde um dos clientes utilizava um fluxo TCP e os demais utilizavam UDP. O próximo experimento avalia o comportamento dos dois algoritmos na presença de outras combinações de padrões de tráfego, além das anteriores. Mais especificamente, os padrões de tráfego utilizados incluem combinações apenas com fluxos UDP, que já não haviam atingido o desempenho esperado na primeira versão, e o impacto de um fluxo UDP intermitente (UDPI), que é interrompido em intervalos de tempo regulares. Esse último fluxo é útil para observar a eficiência da função de retomada de banda do sistema de controle de tráfego: cada vez que o fluxo intermitente é interrompido o sistema deve permitir que os recursos que se tornaram disponíveis sejam realocados a outros fluxos. Para esse experimento, o fluxo intermitente é interrompido e reiniciado a cada 2 segundos.

Para que a vazão possa ser utilizada como fator de comparação quando um cliente utiliza um fluxo UDPI é necessário que pelo menos uma das VMs desse cliente apresente uma demanda constante. Isso ocorre pois só é possível utilizar a vazão como elemento de comparação do grau de correção dos algoritmos durante a ocorrência de congestionamento. No caso do experimento em questão, a demanda constante por uma outra VM do mesmo cliente é necessária para manter a contenção no enlace de acesso quando o fluxo UDPI é interrompido.

A configuração de rede utilizada nesse experimento (três clientes), as garantias de banda de cada cliente e a métrica utilizada são as mesmas do experimento anterior. O posicionamento das VMs foi alterado, sendo que cada cliente possui uma VM em

cada máquina física do ambiente de testes e todos os fluxos de todos os clientes são direcionados para as VMs na máquina 1. Os clientes B e C utilizam sempre o mesmo tipo de padrão de tráfego, que pode ser TCP, UDP ou UDPI, enquanto o cliente A utiliza apenas TCP ou UDP. A figura 4.6 apresenta os resultados. A legenda no eixo x indica a combinação dos padrões de tráfego utilizados pelos clientes A, B e C. A linha superior identifica o tipo fluxo utilizado pelo cliente A; a segunda linha identifica o fluxo utilizado pelos clientes B e C.

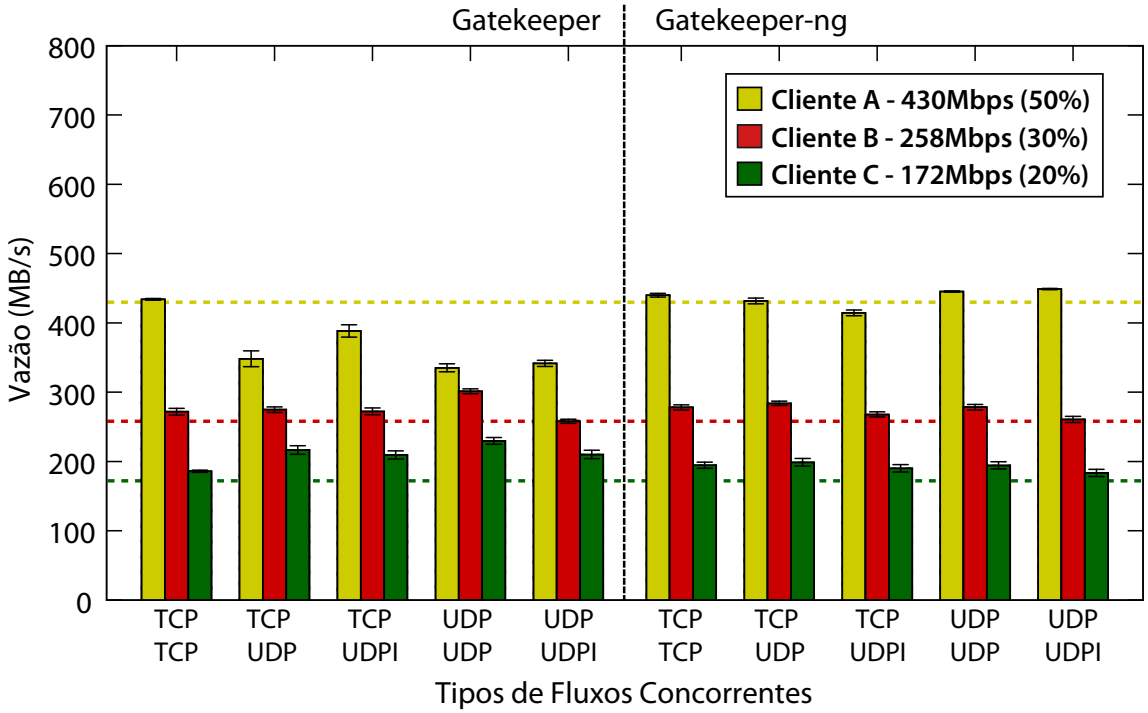


Figura 4.6. Resultados dos Experimentos Comparando os Sistemas com Diferentes Padrões de Tráfego

A partir do gráfico apresentado é possível notar que o Gatekeeper-ng mantém um melhor desempenho que a versão anterior do sistema mesmo na presença de diferentes padrões de tráfego. Na maioria dos casos em que o Gatekeeper-ng é utilizado, as VMs são capazes de atingir uma vazão para a recepção de tráfego que está acima da sua garantia de banda. Esse excesso é o tráfego que "atinge o *headroom*", utilizado pelo Gatekeeper-ng para identificar demandas superiores às garantias de tráfego especificadas. Nesses casos o Gatekeeper-ng irá notificar o problema para as máquinas físicas de onde esse tráfego é originado, mas mesmo assim entregará os dados às VMs. Na versão anterior do sistema, essa porção do tráfego seria descartado pelas filas de entrada, limitando de forma rígida a vazão de recepção das VMs.

Claramente, a versão original do Gatekeeper não é capaz de manter as alocações

definidas para os diversos clientes nesses contextos, exceto para o caso em que todos os clientes utilizam TCP. A maior agressividade dos padrões de tráfego torna o mecanismo de reação pela observação de descartes frequentes menos eficiente para os demais casos, apesar do sistema ainda manter pelo menos uma distribuição de recursos privilegiando o cliente que teria a maior parte da alocação.

4.5 Precisão

Além de avaliar a divisão dos recursos em termos da quantidade total de dados consumidos por cada cliente, a avaliação comparativa buscou analisar a exatidão do controle de tráfego oferecida por cada sistema. A ideia nesse caso é comparar, ao longo do tempo, a divisão de banda realizada pelo sistema de controle de tráfego com a divisão de banda esperada. Para isso, foi realizado um experimento semelhante ao utilizado para comparar o comportamento interno dos dois algoritmos, apresentado na seção 4.2. Durante a ocorrência de congestionamento foram coletadas as estatísticas das interfaces virtuais *vif* das VMs receptoras em intervalos regulares de 20 milissegundos. Esses valores foram comparados aos que deveriam ser obtidos se a divisão de tráfego estivesse exata, isto é, se as garantias fossem respeitadas a todo instante. A diferença entre a alocação de banda ideal e o valor efetivamente observado em cada *vif* é o erro causado por uma divisão de banda incorreta. Esse erro foi utilizado como fator de comparação das duas soluções de controle de tráfego. Os erros não foram computados, porém, nos casos onde essa diferença era positiva, pois redistribuir a banda não utilizada é o comportamento esperado de qualquer escalonador de tráfego com conservação de trabalho.

Nesse experimento foram explorados os erros na alocação de banda considerando alguns dos padrões tráfego utilizados no experimento anterior. O número de máquinas virtuais e de clientes, assim como as garantias de cada um também foram mantidos como naquele caso. A figura 4.7 apresenta a distribuição acumulada (CDF) dos erros calculados segundo o procedimento descrito anteriormente para quatro padrões de tráfego distintos. Assim como no experimento anterior, o primeiro padrão de tráfega identificado na legenda foi utilizado pelo cliente A e o segundo foi utilizado pelos demais.

A partir dos gráficos apresentados na figura 4.7 é possível notar que a precisão do controle de banda oferecida pelo Gatekeeper-ng é superior à oferecida pela versão anterior do sistema. Quando a combinação de fluxos utilizadas é *TCP x UDPI* ou *UDP x UDP*, o Gatekeeper original apresenta um desempenho ruim, sendo que em 60% das

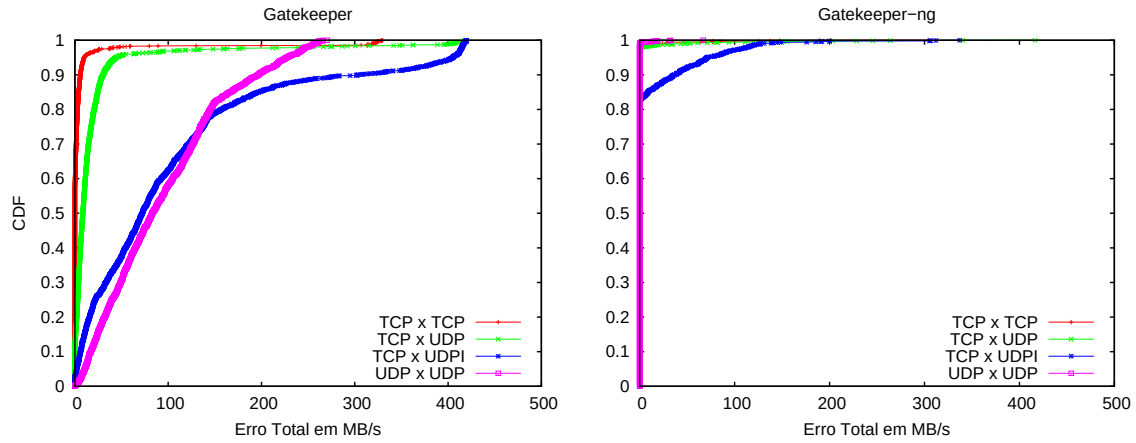


Figura 4.7. Resultados dos Experimentos Comparando a Precisão dos dois Sistemas. Os valores apresentados são os erros na alocação de tráfego das VMs 1/A, 1/B e 1/C combinados.

medições o erro é maior que 100 Mbps. Quando o Gatekeeper-ng é utilizado, apenas para a combinação *TCP x UDPI* há uma ocorrência maior de erros. Mesmo assim, apenas 15% das medições apresentam um erro significativo, e menos de 5% apresentam erro superior a 100 Mbps.

4.6 Sobrecarga de CPU

Os últimos resultados a serem apresentados neste capítulo são relativos ao consumo de CPU das duas soluções. Para avaliar o consumo de CPU foi utilizada a medição do tempo de CPU provida pelo próprio Xen. Como ambos sistemas de controle de tráfego são executado no Dom0, o *overhead* devido a eles se manifesta como aumento do tempo de processamento consumido por aquele domínio. Por esse motivo, utilizamos a monitoração do Xen para coletar o tempo de CPU consumido pelo Dom0 durante cada experimento, que durava 30 segundos.

O gráfico da figura 4.8 apresenta o tempo de CPU observado no experimento que envolve diferentes padrões de tráfego, apresentado na seção 4.4. O gráfico contém o consumo de CPU de três execuções. A primeiro delas, denominada *Sem Controle*, apresenta os resultados da execução do experimento sem nenhum dos sistemas de controle de tráfego ativo. Isto é, o tempo de CPU medido corresponde ao processamento normal do sistema sem o Gatekeeper. Esse custo é devido apenas à operação usual do VMM no escalonamento das VMs e no processamento das operações de entrada-e-saída das mesmas, incluindo o encaminhamento dos pacotes trocados entre as VMs (sem qualquer mecanismo de controle de tráfego nesse nível). A segunda forma de

execução apresenta o consumo de CPU do Dom0 quando o *Gatekeeper-ng* é utilizado para controlar o tráfego de rede e a terceira e última apresenta o consumo de CPU do Dom0 quando a versão original do Gatekeeper é utilizada.

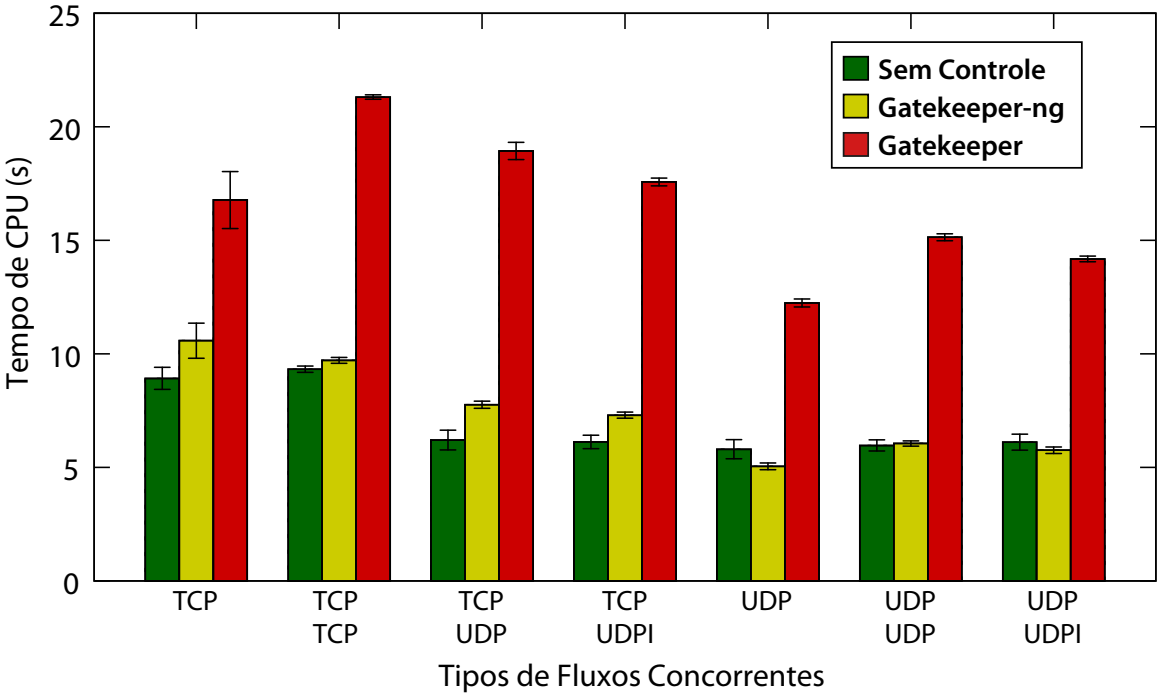


Figura 4.8. Resultados dos Experimentos Comparando os Sistemas com Diferentes Padrões de Tráfego

Os resultados mostram que a sobrecarga de CPU imposta pelo Gatekeeper-ng é pequena, havendo sobreposição dos intervalos de confiança daquela versão com a versão sem controle para muitos casos.

A implementação do Gatekeeper-ng faz uso do fluxo de processamento já otimizado do *Open vSwitch* para computar as taxas de transferência correntes de cada fluxo e só acionar o controlador de congestionamento nos momentos em que a contenção é detectada. Dessa forma, seu custo de processamento por pacote é baixo e a redução da contenção na interface de entrada da máquina pode mesmo reduzir o custo de processamento daquele módulo.

Já a versão original do Gatekeeper, por outro lado, apresenta um consumo de tempo de CPU visivelmente mais elevado para todos os casos. Isso se deve, principalmente, à concentração de responsabilidades no agente de controle de congestionamento daquele sistema, que executa completamente em espaço do usuário e precisa consultar periodicamente estruturas de dados do *kernel* para obter informações sobre descartes, elevando o *overhead* do sistema.

Capítulo 5

Conclusão e Trabalhos Futuros

A tendência de terceirização da infraestrutura computacional alinhada ao crescimento da computação em nuvem aumenta o número clientes que compartilham os recursos de datacenters virtualizados. Paralelamente a isso, provedores de computação-como-serviço buscam maximizar a utilização dos recursos computacionais disponíveis consolidando um número cada vez maior de máquinas virtuais em seus servidores virtualizados. Essas duas tendencias, quando combinadas, aumentam a interação entre diversos clientes no consumo dos recursos compartilhados da infraestrutura do datacenter. Nesse cenário é essencial que o provedor de recursos seja capaz de garantir o isolamento entre máquinas virtuais de diferentes clientes tanto por questões de segurança, quando para atender os requisitos contratuais que especificam o desempenho esperado por cada cliente.

As atuais tecnologias de virtualização oferecem boas soluções para garantir o isolamento de desempenho entre máquinas virtuais em termos de CPU e memória. No entanto, o mesmo não é verdade para os recursos da rede do datacenter. Os mecanismos atuais para o controle de tráfego em datacenters virtualizados não são capazes de garantir um bom isolamento de desempenho entre as máquinas virtuais que compartilham a infraestrutura de rede. Em muitos casos, essas soluções dependem da cooperação das próprias máquinas virtuais, estando vulneráveis à padrões de tráfego "egoístas" que não fazem qualquer esforço para manter uma divisão justa dos recursos de rede.

O Gatekeeper é um sistema de controle de tráfego que busca prover isolamento de desempenho de rede para datacenters virtualizados. A ideia básica do sistema é utilizar a camada de virtualização dos servidores para gerenciar a alocação de banda das máquinas virtuais de acordo com as garantias de tráfego especificadas por cada cliente. Para isso, o Gatekeeper utiliza limitadores de banda associados a cada máquina virtual

que são ajustados dinamicamente quando uma violação das garantias especificadas pelos clientes é detectada. Esse controle é feito de forma distribuída com a participação de cada servidor do datacenter. Nesse trabalho de mestrado o mecanismo de alocação do Gatekeeper foi estudado e aprimorado. Como resultado, foi produzida uma nova versão do sistema, o Gatekeeper-ng.

O Gatekeeper-ng conta com uma nova arquitetura e com novos algoritmos para controle de tráfego desenvolvidos com o objetivo de contornar alguns dos principais pontos fracos da versão original do sistema. A nova versão explora o ambiente virtualizado de maneira mais direta, modificando o subsistema de rede que controla o chaveamento de pacotes da plataforma virtualizada. Com isso foi possível aprimorar a detecção de violações das garantias de tráfego e obter informações mais detalhadas a respeito dos padrões de tráfego envolvidos no problema. Essas informações são utilizadas pelo Gatekeeper-ng para determinar os limites de banda ideais de cada máquina virtual envolvida na violação das garantias. Os experimentos realizados mostram que os ganhos com a nova versão do sistema são significativos, tanto na precisão do controle de tráfego oferecida quanto na redução da sobrecarga computacional imposta pelo uso do sistema.

Em termos de direções para continuação deste trabalho, diversas linhas de ação são possíveis. Um dos próximos passos planejados para o trabalho é a experimentação do Gatekeeper-ng utilizando cargas de trabalho reais em um ambiente de testes de maiores proporções. Experimentos no ambiente Open Cirrus[1], que dispõe de um grande número de servidores compartilhados entre diversos grupos de pesquisa, estão em fase de planejamento. Além disso, também pretende-se aprimorar o mecanismo de retomada de banda do Gatekeeper-ng, que é ativado quando o sistema deixa de detectar contenção, que atualmente é baseado em uma função de incremento linear. Mecanismos de retomada de banda baseados em funções quadráticas ou cúbicas, como o utilizado pelo CUBIC-TCP, estão em fase de testes. Considerando o aumento do número VMs suportadas por cada servidor virtualizado, uma outra demanda é a extensão do Gatekeeper-ng para controlar o tráfego de rede interno às máquinas físicas, já que no momento toda a arquitetura é focada no tráfego recebido na interface física do servidor. Estender o mecanismo para todas as interfaces virtuais pode exigir a implementação de algoritmos distribuídos para lidar com o estado de conexões que atravessem cada interface — ou combinação delas. Em outra linha de investigação, seria interessante explorar outros benefícios da integração do Gatekeeper-ng ao arcabouço *OpenFlow/Open vSwitch*. Esses sistemas foram desenvolvidos para aumentar a

[1] https://opencirrus.org/

flexibilidade de programação e configuração da rede, no contexto de redes definidas por software (*software defined networks*). Utilizando esse conceito é possível, por exemplo, programar as *Open vSwitches* de um datacenter para que elas ofereçam para os clientes a concretização da abstração do *switch* único conectando todas as VMs de um cliente. A integração desses conceitos pode gerar ferramentas de gerência e configuração de redes de datacenters com recursos ainda não disponíveis nas soluções atuais.

Referências Bibliográficas

[1] Al-Fares, M.; Loukissas, A. & Vahdat, A. (2008). A scalable, commodity data center network architecture. In *Proceedings of the ACM SIGCOMM 2008 Conference*, pp. 63--74, Seattle, WA.

[2] Alizadeh, M.; Atikoglu, B.; A. Kabbani, A. L.; Pan, R.; Prabhakar, B. & Seaman, M. (2008). Data center transport mechanisms: congestion control theory and ieee standardization. In *Proceedings of the 46th Annual Allerton Conference on Communications, Control and Computing*, Monticello, Illinois, USA. IEEE Computer Society.

[3] Allman, M.; Paxson, V. & Blanton, E. (2009). RFC 5681: Tcp congestion control.

[4] Arlitt, M. & Jin, T. (99). Workload characterization of the 1998 world cup web site. Relatório Técnico HPL-1999-35R1, Hewlett-Packard Laboratories.

[5] Armbrust, M.; Fox, A.; Griffith, R.; Joseph, A. D.; Katz, R. H.; Konwinski, A.; Lee, G.; Patterson, D. A.; Rabkin, A.; Stoica, I. & Zaharia, M. (2009). Above the clouds: A berkeley view of cloud computing. Technical Report UCB/EECS-2009-28, EECS Department, University of California, Berkeley.

[6] Arregoces, M. & Portolani, M. (2003). *Data Center Fundamentals: Understand Data Canter network design and infrastructure architecture, including load balancing, SSL, and security.* Cisco Press, 1st edição.

[7] Barham, P.; Dragovic, B.; Fraser, K.; Hand, S.; Harris, T.; Ho, A.; Neugebauer, R.; Pratt, I. & Warfield, A. (2003). Xen and the art of virtualization. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, pp. 164--177.

[8] Benson, T.; Akella, A. & Maltz, D. A. (2010). Network traffic characteristics of data centers in the wild. In *Proceedings of the 10th annual conference on Internet measurement*, IMC '10, pp. 267--280, New York, NY, USA. ACM.

[9] Bitbucket (2009). Bitbucket: On our extended downtime, amazon and what's coming. http://blog.bitbucket.org/2009/10/04/on-our-extended-downtime-amazon-and-whats-coming/.

[10] Braden, R.; Clark, D. & Shenker, S. (1994). RFC 1633: integrated services in the internet architecture: an overview.

[11] Chesire, M.; Wolman, A.; Voelker, G. M. & Levy, H. M. (01). Measurement and analysis of a streaming-media workload. In *USITS'01: Proceedings of the 3rd Conference on USENIX Symposium on Internet Technologies and Systems*, pp. 1--1, Berkeley, CA, EUA. USENIX Association.

[12] Cisco Systems, I. (2007). Cisco data center infrastructure design guide 2.5.

[13] Citrix XenDesktop (2010). http://www.citrix.com/virtualization/desktop/xendesktop.html.

[14] Corbet, J.; Rubini, A. & Kroah-Hartman, G. (2005). *Linux Device Drivers, 3rd Edition*. O'Reilly Media, Inc.

[15] Dean, J. & Ghemawat, S. (2004). Mapreduce: Simplified data processing on large clusters. In *Proceedings of the 6th Symposium on Operating System Design and Implementation (OSDI '04)*, pp. 137--150, San Francisco, CA.

[16] Devera, M. (2011). Hierarchical token bucket theory. Acessado em 9 de Junho de 2011.

[17] Duffield, N. G.; Goyal, P.; Greenberg, A.; Mishra, P.; Ramakrishnan, K. K. & van der Merive, J. E. (1999). A flexible model for resource management in virtual private networks. In *Proceedings of the ACM SIGCOMM 1999 Conference*, pp. 95--108, New York, NY, USA. ACM.

[18] EC2 (2010). Amazon Elastic Compute Cloud. http://aws.amazon.com/ec2/.

[19] Greenberg, A.; Hamilton, J.; Maltz, D. A. & Patel, P. (2008a). The cost of a cloud: research problems in data center networks. *SIGCOMM Comput. Commun. Rev.*, 39:68--73.

[20] Greenberg, A.; Hamilton, J. R.; Jain, N.; Kandula, S.; Kim, C.; Lahiri, P.; Maltz, D. A.; Patel, P. & Sengupta, S. (2009). VL2: A scalable and flexible data center network. In *Proceedings of the ACM SIGCOMM 2009 Conference*, pp. 51--62, Spain.

[21] Greenberg, A.; Lahiri, P.; Maltz, D. A.; Patel, P. & Sengupta, S. (2008b). Towards a next generation data center architecture: Scalability and commoditization. In *Proceedings of the ACM workshop on Programmable Routers for Extensible Services of Tomorrow (PRESTO '08)*, pp. 57--62, Seattle, WA.

[22] Guo, C.; Lu, G.; Li, D.; Wu, H.; Zhang, X.; Yunfeng Shi, C. T.; Zhang, Y. & Lu, S. (2009). BCube: A high performance, server-centric network architecture for modular data centers. In *Proceedings of the ACM SIGCOMM 2009 Conference*, pp. 63--74, Barcelona, Spain.

[23] Guo, C.; Lu, G.; Wang, H. J.; Yang, S.; Kong, C.; Sun, P.; Wu, W. & Zhang, Y. (2010). Secondnet: a data center network virtualization architecture with bandwidth guarantees. In *Proceedings of the 6th International COnference*, Co-NEXT '10, pp. 15:1--15:12, New York, NY, USA. ACM.

[24] Ha, S.; Rhee, I. & Xu, L. (2008). Cubic: a new tcp-friendly high-speed tcp variant. *SIGOPS Oper. Syst. Rev.*, 42:64--74.

[25] Hyser, C.; Mckee, B.; Gardner, R. & Watson, B. J. (07). Autonomic virtual machine placement in the data center. Relatório Técnico HPL-2007-189, Hewlett-Packard Laboratories.

[26] IBM VM/370 (2011). Virtual machine operating system history and heritage. Acessado em 9 de Julho de 2011.

[27] Ioannidis, J. & Bellovin, S. M. (2002). Implementing pushback: Router-based defense against ddos attacks. In *Proceedings of the Network and Distributed System Security Symposium*, San Diego, CA.

[28] Jackson, K. R.; Ramakrishnan, L.; Runge, K. J. & Thomas, R. C. (2010). Seeking supernovae in the clouds: a performance study. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, pp. 421--429, New York, NY, USA. ACM.

[29] Jacobson, V. (1988). Congestion avoidance and control. *SIGCOMM Comput. Commun. Rev.*, 18:314--329.

[30] Jain, R. K. (91). *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling,*. Wiley-Interscience, New York, NY, EUA.

[31] Kabbani, A.; Alizadeh, M.; Yasuda, M.; Pan, R. & Prabhakar, B. (2010). Af-qcn: Approximate fairness with quantized congestion notification for multi-tenanted data centers. In *Proceedings of the 2010 18th IEEE Symposium on High Performance Interconnects*, HOTI '10, pp. 58--65, Washington, DC, USA. IEEE Computer Society.

[32] Kandula, S.; Sengupta, S.; Greenberg, A.; Patel, P. & Chaiken, R. (2009). The nature of data center traffic: measurements & analysis. In *Proceedings of the 9th ACM SIGCOMM conference on Internet measurement conference*, IMC '09, pp. 202--208, New York, NY, USA. ACM.

[33] Kangarlou, A.; Gamage, S.; Kompella, R. R. & Xu, D. (2010). vsnoop: Improving tcp throughput in virtualized environments via acknowledgement offload. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pp. 1--11, Washington, DC, USA. IEEE Computer Society.

[34] Lam, T.; Radhakrishnan, S.; Vahdat, A. & Varghese, G. (2010). Netshare: Virtualizing data center networks across services. Technical report, University of California, San Diego.

[35] Lee, G.; Tolia, N.; Ranganathan, P. & Katz, R. H. (2009). A case for topology-aware resource allocation for data-intensive applications in the cloud. Technical Report HPL-2009-335, HP Labs, Palo Alto, CA.

[36] Love, R. (2005). *Linux Kernel Development (2nd Edition) (Novell Press)*. Novell Press.

[37] McKeown, N.; Anderson, T.; Balakrishnan, H.; Parulkar, G. M.; Peterson, L. L.; Rexford, J.; Shenker, S. & Turner, J. S. (2008). Openflow: enabling innovation in campus networks. *Computer Communication Review*, 38(2):69--74.

[38] Meng, X.; Pappas, V. & Zhang, L. (2010). Improving the scalability of data center networks with traffic-aware virtual machine placement. In *Proceedings of the 29th conference on Information communications*, INFOCOM'10, pp. 1154--1162, Piscataway, NJ, USA. IEEE Press.

[39] Microsoft Hyper-V (2010). `http://www.microsoft.com/hyper-v-server`.

[40] Mills, D.; Martin, J.; Ed.; Burbank, J. & Kasch, W. (2010). *Network Time Protocol Version 4: Protocol and Algorithms Specification*. Internet Engineering Task Force. RFC 5905.

[41] Mudigonda, J.; Yalagandula, P.; Al-Fares, M. & Mogul, J. C. (2010). Spain: Cots data-center ethernet for multipathing over arbitrary topologies. In *Proceedings of the 7th Symposium on Networked Systems Design and Implementation (NSDI)*, San Jose, CA.

[42] Mysore, R. N.; Pamboris, A.; Farrington, N.; Huang, N.; Pardis Miri, S. R.; Subramanya, V. & Vahdat, A. (2009). PortLand: A scalable fault-tolerant layer 2 data center network fabric. In *Proceedings of the ACM SIGCOMM 2009 Conference*, Barcelona, Spain.

[43] Nanda, S. & cker Chiueh, T. (2005). A survey of virtualization technologies. Technical report, Department of Computer Science, SUNY at Stony Brook.

[44] Nurmi, D.; Wolski, R.; Grzegorczyk, C.; Obertelli, G.; Soman, S.; Youseff, L. & Zagorodnov, D. (08). The eucalyptus open-source cloud-computing system. In *CAA'08: Proceedings of the 1st Workshop on Cloud Computing and Its Applications*.

[45] Peterson, L. L. & Davie, B. S. (2007). *Computer Networks: A Systems Approach, Fourth Edition (The Morgan Kaufmann Series in Networking)*. Morgan Kaufmann, 4 edição.

[46] Pfaff, B.; Pettit, J.; Koponen, T.; Amidon, K.; Casado, M. & Shenker, S. (2009). Extending Networking into the Virtualization Layer. In *Proceedings of the 8th Workshop on Hot Topics in Networks (HotNets)*, New York, NY.

[47] Raghavan, B.; Vishwanath, K. V.; Ramabhadran, S.; Yocum, K. & Snoeren, A. C. (2007). Cloud control with distributed rate limiting. In *Proceedings of the ACM SIGCOMM 2007 Conference*, pp. 337--348, Kyoto, Japan.

[48] Rodrigues, H.; Santos, J. R.; Turner, Y.; Soares, P. & Guedes, D. (2011). Gatekeeper: supporting bandwidth guarantees for multi-tenant datacenter networks. In *Proceedings of the 3rd conference on I/O virtualization*, WIOV'11, pp. 6--6, Berkeley, CA, USA. USENIX Association.

[49] Salim, J.; Khosravi, H.; Kleen, A. & Kuznetsov, A. (2003). Linux Netlink as an IP Services Protocol. RFC 3549 (Informational).

[50] Schlansker, M.; Tourrilhes, J.; Turner, Y. & Santos, J. R. (2010). Killer fabrics for scalable datacenters. In *IEEE International Conference on Communications (ICC)*.

[51] Shi, W.; Wright, R.; Collins, E. & Karamcheti, V. (02). Workload characterization of a personalized web site - and its implications for dynamic content caching. In

WCW'02: Proceedings of the 7th International Conference on Web Content Caching and Distribution, pp. 1–16.

[52] Shieh, A.; Kandula, S.; Greenberg, A. & Kim, C. (2010). Seawall: Performance isolation for cloud datacenter networks. In *Proceedings of the Workshop on Hot Topics in Cloud Computing*, Boston, MA, USA.

[53] Shieh, A.; Kandula, S.; Greenberg, A. & Kim, C. (2011). Sharing the data center network. In *Proceedings of the 8th Symposium on Networked Systems Design and Implementation (NSDI)*, Boston, MA, USA.

[54] Soares, P. V. (2010). Gatekeeper: Controle de tráfego distribuído em datacenters virtualizados. Dissertação de Mestrado.

[55] Soares, P. V.; Santos, J. R.; Turner, Y.; Tolia, N. & Guedes, D. (2010). Gatekeeper: Distributed rate control for virtualized datacenters. Technical Report HPL-2010-151, HP Laboratories.

[56] Stoica, I.; Shenker, S. & Zhang, H. (2003). Core-stateless fair queueing: a scalable architecture to approximate fair bandwidth allocations in high-speed networks. *IEEE/ACM Transactions on Networks*, 11(1):33--46.

[57] Sundararaj, A. I.; Gupta, A. & Dinda, P. A. (04). Dynamic topology adaptation of virtual networks of virtual machines. In *LCR '04: Proceedings of the 7th Workshop on languages, compilers, and run-time support for scalable systems*, pp. 1--8, New York, NY, EUA. ACM.

[58] Sundararaj, A. I.; Sanghi, M.; Lange, J. R. & Dinda, P. A. (06). Hardness of approximation and greedy algorithms for the adaptation problem in virtual environments. In *ICAC '06: Proceedings of the 2006 IEEE International Conference on Autonomic Computing*, pp. 291--292, Washington, DC, EUA. IEEE Computer Society.

[59] Vaquero, L. M.; Rodero-Merino, L.; Caceres, J. & Lindner, M. (2008). A break in the clouds: towards a cloud definition. *SIGCOMM Comput. Commun. Rev.*, 39:50--55.

[60] Veloso, E.; Almeida, V.; Meira, Jr., W.; Bestavros, A. & Jin, S. (06). A hierarchical characterization of a live streaming media workload. *IMC'06: Proceedings of the 2nd ACM SIGCOMM Workshop on Internet measurment*, 14(1):133--146.

[61] Virtuoso (08). `http://virtuoso.cs.northwestern.edu`, acessado em 06 de novembro de 2009.

[62] VMware (2010). ESX server 3 configuration guide. `http://www.vmware.com/pdf/vi3_35/esx_3/r35/vi3_35_25_3_server_config.pdf`.

[63] VMWare ESX Server (2011). `http://www.vmware.com`, acessado em 04 de junho de 2011.

[64] Wang, G. & Ng, T. S. E. (2010). The impact of virtualization on network performance of amazon ec2 data center. In *Proceedings of the 29th conference on Information communications*, INFOCOM'10, pp. 1163--1171, Piscataway, NJ, USA. IEEE Press.

[65] Wang, Q.; Makaroff, D.; Edwards, H. K. & Thompson, R. (03). Workload characterization for an e-commerce web site. In *CASCON '03: Proceedings of the 2003 Conference of the Centre for Advanced Studies on Collaborative research*, pp. 313--327. IBM Press.

[66] Whitaker, A.; Shaw, M. & Gribble, S. D. (2002). Denali: Lightweight virtual machines for distributed and networked application. Technical report, University of Washington.

[67] Wine (2011). Wine project. Acessado em 1 de julho de 2011.

[68] Wood, T.; Shenoy, P. J.; Venkataramani, A. & Yousif, M. S. (07). Black-box and gray-box strategies for virtual machine migration. In *NSDI'07: Proceedings of the 4th USENIX Symposium on Networked Systems Design and Implementation*, Cambridge, MA, EUA.

[69] Zhang, Q.; Cheng, L. & Boutaba, R. (2010). Cloud computing: state-of-the-art and research challenges. *Journal of Internet Services and Applications*, 1:7–18. 10.1007/s13174-010-0007-6.

[70] Zhu, X.; Young, D.; Watson, B. J.; Wang, Z.; Rolia, J.; Singhal, S.; Mckee, B.; Hyser, C.; Gmach, D.; Gardner, R.; Christian, T. & Cherkasova, L. (09). 1000 islands: an integrated approach to resource management for virtualized data centers. *Journal of Cluster Computing*, 12(1):45--57.

Apêndice

Apêndice A

Observações sobre programação no *kernel* linux

O desenvolvimento de código que executa junto ao Kernel Linux deve ser tratado como código do próprio sistema operacional, obedecendo todas as particularidades do sistema. Embora a tarefa de escrever códigos para um sistema operacional não seja necessariamente mais difícil, ela possui algumas característica que a distinguem da implementação de uma aplicação comum executada a nível de usuário[36]. Algumas dessas diferenças são:

- Inexistência de *libc*: Todo programa implementado a nível de usuário possui acesso à biblioteca padrão do C (*C Standard Library — libc*), que provê muitas das funcionalidades comumente utilizadas por diversas aplicações. O núcleo do Linux, no entanto, não inclui os códigos que compõem a libc, o aumento do tamanho do binário contendo o SO e a parca de desempenho as principais razões para isso. Ao invés de incluir a libc, o Kernel Linux possui implementações próprias de algumas funcionalidades, como manipulação de strings e controle de fluxos de execução concorrentes (*threads*).

- Gerencia de Memória: Diferentemente de uma aplicação que executa a nível de usuário, o Kernel não possui um sistema que gerencia o seu acesso à memória. Com a possibilidade acessar qualquer endereço de memória, alterações em posições impróprias podem levar a um problema irreversível — ao invés de apenas causar a emissão de um sinal indicando uma falha de segmentação (*segmentation fault*). Além disso, o Kernel não faz paginação da memória que utiliza: cada byte de memória consumido pelo Kernel é um byte a menos disponível na memória física.

Uma outra diferença é o tamanho da pilha. Enquanto uma aplicação a nível de usuário possui uma pilha que pode crescer dinamicamente, o Kernel possui uma pequena e de tamanho fixo, geralmente de 8KB em arquiteturas de 32 bits e 16KB em arquiteturas de 64 bits. Isso significa que os algoritmos implementados no Kernel não podem se dar ao luxo de utilizar uma árvore de recursão grande ou qualquer técnica que precise de muito espaço da pilha de recursão.

- Falta de ponto flutuante: Por questões de desempenho, o Kernel não oferece suporte a operações de ponto flutuante.

  Para entender o porque dessa característica, basta conhecer o procedimento realizado pelo Kernel ao gerenciar o acesso ao processador de aplicações a nível de usuário. Toda vez que ocorre uma *troca de contexto* quando uma aplicação está sendo executada a nível de usuário, o Kernel retoma o controle do processador e volta à execução. Porém, antes de qualquer coisa, o SO precisa salvar o conteúdo dos registradores do processador para que, quando o processo que estava executando retome o processador, o conteúdo dos registradores possa ser restaurado. Com isso, ao retomar novamente o processador, o processo a nível de usuário nem sequer percebe que houve uma pausa em sua execução.

  Muitos processadores possuem conjuntos de registradores distintos para operações entre números inteiros e operações de ponto flutuante. Como a cópia do conteúdo dos registradores impõe uma certa sobrecarga na execução do sistema operacional, os desenvolvedores do Kernel Linux decidiram por abolir o uso de ponto flutuante do Kernel.

  Essa é uma característica diretamente ligada à implementação do Gatekeeper-ng, pois os cálculos de utilização de banda dependem diretamente de operações que envolvem números não inteiros.

- Sincronização e condições de corrida: O Kernel Linux é um sistema operacional preemptivo e multitarefas. Por conta disso, fluxos de execução são constantemente inseridos e removidos do processador de acordo com o escalonador de processos do Linux. Isso significa que a implementação do SO deve se preocupar constantemente com a sincronização do acesso às variáveis compartilhadas para evitar inconsistências. Como o Kernel também é capaz de ser executado em sistemas multiprocessados, a falta de cuidado com o acesso às variáveis compartilhadas pode fazer com que um mesmo código sendo executado em diferentes processadores acesse a mesma variável. Além disso, esse ambiente está propenso

ao surgimento de diversas condições de corrida, que devem ser tratadas pelo programador.

O Kernel é um ambiente de programação incomum: sem proteção de memória, ausência de libc, pilha pequena e de tamanho fixo, um ambiente altamente concorrente e com uma grande quantidade de códigos fonte desenvolvidos por diversos programadores de todo o mundo. Porém, apesar de todas essas particularidades, o núcleo do sistema operacional não deixa de ser apenas um programa com características diferenciadas e que exigem maior atenção e conhecimentos específicos por parte do programador.

Apêndice B

Manipulação de tempo no Linux

A informação de tempo disponibilizada pelo Kernel Linux depende dos componentes de hardware instalados na máquina física que registram o passar do tempo utilizando interrupções de hardware periódicas. Por esse motivo, a precisão do relógio do sistema pode diferir de um sistema para outro, sendo que alguns são capazes de registrar intervalos de tempo ordens de magnitude menores que outros.

No Kernel Linux as duas principais variáveis globais que mantêm a informação sobre o Relógio do Sistema são `HZ` e `jiffies`. A variável `jiffies` é atualizada a cada medição de tempo pelo *Relógio do Sistema* (*System Timer*). A variável `HZ` é uma variável estática que indica quantos `jiffies` contém 1 segundo. Logo, se `HZ` = 100, então a diferença de tempo entre duas atualizações consecutivas da variável `jiffies` é de 10 milissegundos. A abstração do relógio do sistema composta por essas duas variáveis foi adotada pelo Kernel Linux por questões de portabilidade. O código que utiliza apenas essas variáveis como fonte de tempo pode ser executado em diferentes arquiteturas, pois é o Linux que cuida de tratar as interrupções de hardware relativas ao Relógio do Sistema de diferentes arquiteturas e atualizar as variáveis `HZ` e `jiffies`.

A frequência de interrupções geradas pelo Relógio do Sistema, armazenada na variável `HZ`, é de 100 interrupções por segundo. Esse é o valor padrão do Kernel Linux e pode ser alterado antes da compilação do sistema operacional. O valor máximo para a variável `HZ` suportado pelo Linux é de 1000 interrupções por segundo, mas essa maior granularidade do Relógio está condicionado à disponibilidade de um componente de hardware que seja capaz de operar à essa frequência. Obviamente quanto maior a frequência de interrupções, maior a sobrecarga do sistema, que deve tratar as interrupções e também colocar em execução os códigos sistema que são atrelados ao tique-taque do Relógio do Sistema.

Além do relógio do sistema, um outro conceito muito importante, e que é impres-

cindível para o desenvolvimento de código a ser executado junto ao subsistema de rede do Linux, são os *Socket Buffers*, codificados na estrutura `sk_buff`. Todo dado a ser manipulado pelo subsistema de rede do Linux é encapsulado em uma dessas estruturas, que representam um pacote de rede. O Kernel contém diversas funções para manipular o conteúdo de um *Socket Buffer* (*SKB*), dentre elas alocar espaço em memória, calcular *checksums*, adicionar cabeçalhos, etc..

Dentre os diversos campos da estrutura `sk_buff`, a implementação realizada nesse trabalho faz uso frequente de dois deles, o `sk_buff.len` e o `sk_buff.cb[]`. O primeiro deles armazena o tamanho do pacote encapsulado na estrutura e o segundo é um vetor que pode ser considerado uma "área de memória livre", presente em cada *Socket Buffer*. O nome dado a essa área de memória, `cb`, é proveniente da abreviação de *Control Buffer*, pois ela é destinada à implementações de protocolos de rede que precisam manter alguma informação de controle em cada pacote. O uso desse vetor por parte da implementação realizada é descrita em detalhes após a apresentação da implementação do monitor de tráfego.