GATEKEEPER: CONTROLE DE TRÁFEGO DISTRIBUÍDO EM DATACENTERS VIRTUALIZADOS

PAOLO VICTOR GONÇALVES SOARES

GATEKEEPER: CONTROLE DE TRÁFEGO DISTRIBUÍDO EM 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

Maio de 2010

Gonçalves Soares, Paolo Victor
D1234p Gatekeeper: Controle de Tráfego Distribuído em Datacenters Virtualizados / Paolo Victor Gonçalves Soares. — Belo Horizonte, 2010
xiv, 67 f. : il. ; 29cm

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

1. Redes. 2. Sistemas Distribuídos. 3. QoS. 4. Cloud Computing. I. Ttulo.

CDU 519.6*82.10

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

FOLHA DE APROVAÇÃO

Gatekeeper: Controle de banda distribuído em ambientes virtualizados

PAOLO VICTOR GONÇALVES SOARES

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. VIRGÍLIO AUGUSTO FERNANDES ALMEIDA
Departamento de Ciência da Computação - UFMG

PROF. WAGNER MEIRA JÚNIOR
Departamento de Ciência da Computação - UFMG

Belo Horizonte, 31 de maio de 2010.

À minha família, por seu amor e apoio incondicionais.

Aos meus amigos, por aturarem meu pessimismo constante e me lembrarem de que há um mundo bacana lá fora.

Aos meus colegas, pelo auxílio imprescindível para transpor esta barreira.

Sem vocês, este trabalho não passaria de páginas em branco.

Agradecimentos

Agradeço...

À Painho, Mainha e Elaine, meu porto seguro neste mundo;

À Clarísia e Naide, pelo apoio incondicional em todos os momentos;

À Flavio, Eduardo e Celso, companheiros desde os saudosos tempos do LSD;

À Mandinha, Lu/Ju/Lu, Lívia, Kissia, Manu, Rebeca, Seu João e Vânia, os melhores presentes que BH me deu;

À Átila, Obama, Pepe e Pituxa, os melhores amigos de quatro patas que encontrei em BH;

À Macambira, Hélio, Henrique e tantos outros companheiros de luta do Speed;

À Dorgival, por sua orientação e paciência com meus frequentes pings;

À Renato, Yoshio e Niraj, pelo apoio acadêmico e por sua hospitalidade.

"Qualquer tecnologia suficientemente avançada é indistinguível de mágica."
(Arthur C. Clarke)

Resumo

Um datacenter abriga diversos serviços que compartilham uma estrutura física. A divisão de recursos computacionais destes serviços é geralmente feita por tecnologias de virtualização, que criam coleções de máquinas virtuais para cada serviço. Em modelos como *Cloud computing* é esperado que serviços pertencentes a múltiplos usuários compartilhem os mesmos recursos físicos dos servidores do *datacenter*, especificamente: processadores, memória primária e acesso à infra-estrutura de rede. Isto implica na necessidade de que a camada de virtualização ofereça um isolamento eficiente de recursos entre máquinas virtuais.

Embora as tecnologias de virtualização atuais suportem o isolamento de recursos de processamento e memória, o mesmo não pode ser afirmado sobre o acesso à infra-estrutura de rede, dado que as tecnologias atuais se restringem a garantir taxas de envio de dados, provendo garantias de recepção apenas quando o tráfego de rede implementa alguma forma de controle de fluxo, ou na presença de elementos de *hardware* de malha de rede específicos. Com a consolidação de arquiteturas de rede de larga escala, baseadas em *hardware* simples e com alta redundância, o gargalo da rede deixa de estar nos elementos que formam a malha, se concentrando nos enlaces individuais dos servidores. Esta mudança no gargalo de rede aponta a necessidade de um mecanismo de controle de tráfego que seja executado pelos servidores mas que não sofra as limitações inerentes às tecnologias atuais,

Este trabalho apresenta o *Gatekeeper*, um mecanismo distribuído de controle de tráfego, que opera na camada de virtualização. O *Gatekeeper* é transparente às máquinas virtuais executadas em servidor, oferecendo controle de tráfego de transmissão e recepção para cada máquina virtual, além de detecção e resolução de congestionamentos nos enlaces de entrada dos servidores.

Palavras-chave: Sistemas Distribuídos, Virtualização, Cloud Computing, Controle de Tráfego de Rede.

Abstract

A datacenter hosts multiple services that share a single physical infrastructure. On these environments, the sharing of computational resources among the different services is implemented by virtualization technologies, that create collections of virtual machines for each service. In the Cloud Computing model, it's expected that services that belong to different users end up sharing the same physical resources, including processors, memory and network access. This urges the implementation of proper resource sharing and isolation between these virtual machines, for both security and efficiency reasons.

Even though the current virtualization technologies support the isolation of processing and memory resources, the current network sharing technologies, that control the transmission and reception rates of the virtual machines, are only reliable if the virtual machines' traffic implements some form of flow control algorithm, like TCP traffic, or if certain hardware elements are modified or added to the network trunk. The consolidation of high-redundancy, high-performance network architectures that can be built with simpler hardware suggests that a server-side solution that overcomes the current technologies' limitations is needed.

This work presents *Gatekeeper*, a distributed traffic control mechanism that operates on the virtualization layer. *Gatekeeper* is transparent to the virtual machines hosted at the datacenter, and offers both transmission and reception rate guarantees for each virtual machine, the later being ensured by a distributed network congestion detection and mitigation algorithm.

Keywords: Distributed Systems Virtualization, Cloud Computing, Network Traffic Control.

Lista de Figuras

Lista de Tabelas

Sumário

Capítulo 1

Introdução

A área de *Cloud computing* é uma área emergente na computação, que afeta diretamente a forma com a qual são abordados a infraestrutura de TI, os serviços de rede e as aplicações. O termo *Cloud computing* pode ser definido como um modelo que possibilita o acesso através da rede a um conjunto de recursos compartilhados. Esse acesso é feito de forma simples e sob demanda, sendo que os recursos acessados podem ser providos e liberados com o mínimo de intervenção gerencial. Entre as principais características presentes em serviços de *Cloud computing*, estão a sua elasticidade, ou a possibilidade de expandir ou reduzir a quantidade de recursos utilizados de acordo com os requisitos específicos dos clientes; o modelo de pagamento proporcional, no qual o cliente só paga pelos serviços quando ele os utiliza; a maior resiliência, pois falhas nos serviços são isoladas dos usuários e gerenciadas automaticamente pelos provedores e pelo compartilhamento de recursos da infraestrutura por múltiplos clientes.

Essencialmente, o atrativo desse modelo é a forma com a qual ele terceiriza a maioria dos custos de construção de infraestrutura inerentes ao desenvolvimento de aplicações de larga escala, como a compra e instalação de servidores, gerenciamento de rede, administração do espaço físico etc. Os serviços de *Cloud computing* são mais flexíveis comparados aos serviços tradicionais de hospedagem, por automatizar o controle de falhas e a alocação e gerência de recursos. Estas vantagens, aliadas à massificação das aplicações puramente *web*, fazem com que esse modelo deixe de fazer parte apenas do universo acadêmico, se expandindo rapidamente no meio comercial [4, 45, 15].

Como mencionado no parágrafo acima, uma das características do modelo de *Cloud computing* é que os clientes compartilham os recursos de uma infraestrutura gerenciada pelo provedor do serviço. Por questões de privacidade, segurança e desempenho, o provedor de serviço deve isolar os recursos alocados entre os clientes,

garantindo que um cliente não possa acessar dados ou prejudicar o desempenho do serviço prestado aos outros clientes do mesmo serviço. A forma com a qual os recursos são isolados depende da natureza do serviço prestado, por exemplo: em um serviço como o *Google AppEngine* [18], o serviço é acessado através de uma *API*[1], disponibilizada pelo provedor de serviço. Neste caso, como o cliente não tem nenhum controle sobre os servidores que executarão o serviço, o provedor de serviço pode usar mecanismos simples de escalonamento e controle de acesso para implementar o isolamento necessário. No outro extremo do espectro há serviços como o *Amazon EC2* [15] e o *Rackspace* [45], que permitem que os clientes acessem máquinas que executam sistemas operacionais completos, o que oferece um alto grau de flexibilidade e ao mesmo tempo requer a aplicação de mecanismos de isolamento de recursos mais complexos e robustos. A solução consolidada para estes casos é a utilização de ambientes virtualizados.

Em um ambiente virtualizado, cada servidor físico que forma a infraestrutura pode executar uma ou mais máquinas virtuais, cada uma oferecendo as mesmas funcionalidades de uma máquina física. Uma máquina virtual tem um comportamento idêntico ao de uma máquina física, com capacidades garantidas de processamento, memória e E/S. Cabe ao sistema de virtualização gerenciar os recursos físicos dos servidores, dividindo-os entre as máquinas virtuais para, primeiramente, isolar os recursos entre as máquinas virtuais que ocupam os mesmos servidores físicos e também cumprir as garantias de capacidade contratadas pelos clientes. As vantagens do uso de ambientes virtualizados incluem a maior eficácia no uso dos recursos da infraestrutura, a transparência para as aplicações executadas pelos clientes, que não precisam ser modificadas para serem executadas em um ambiente virtualizado e a aplicação de mecanismos de migração, que permitem transferir máquinas virtuais entre diferentes servidores que, por sua vez, podem ser utilizados para aumentar a resiliência dos serviços.

Apesar da eficácia atingida pelo uso de ambientes virtualizados, a popularização do modelo de *Cloud Computing* e o subsequente aumento da carga imposta aos *datacenters* que formam as infraestruturas dos provedores de serviços já começa, em alguns casos, a causar problemas como a degradação de desempenho e latência elevada em períodos de pico de utilização, em parte devido à sobrecargas na rede [42, 53]. Parte dos esforços para amenizar esse efeito se concentram na elaboração de novas tecnologias e arquiteturas da malha de rede [20, 25, 24, 36, 1, 37, 47, 52] de alto desempenho escaláveis e no desenvolvimento de sistemas de gerência autonômica [56, 33, 40, 39], que focam na otimização do uso de recursos dos *datacenters*, realocando máquinas virtu-

[1] Application Programming Interface

ais entre diferentes servidores ou explorando caminhos de comunicação alternativos na malha, com o objetivo final de honrar as garantias de disponibilidade, processamento, memória e latência de comunicação definidas nos contratos de serviço, utilizando o mínimo de recursos do *datacenter*.

1.1 Motivação

No caso provedores de *Cloud computing* que disponibilizam os recursos na forma de máquinas virtuais, o contrato de serviço especifica as capacidades de processamento, memória e E/S das máquinas virtuais alocadas para cada cliente. Especificamente, os clientes podem escolher o processador, quantidade de memória principal e a taxa garantida para transmissões pela rede. Uma omissão visível nesta lista de opções é a opção de definir uma taxa garantida mínima para recepção pela rede, ao invés de transmissão. A ausência desta opção se dá tanto porque garantir simultaneamente as taxas de transmissão e a recepção são objetivos conflitantes (por exemplo: no caso de múltiplas transmissões para um mesmo receptor, como agir quando a soma dos limites de múltiplos transmissores é maior que o limite de recepção?), quanto por características das tecnologias de controle de tráfego disponíveis, que ora possuem custos proibitivos ou dependem do padrão de comunicação de rede das aplicações para oferecer um controle confiável. Antes de detalhar esta dependência, é necessário especificar as alternativas de controle de tráfego disponíveis no ambiente em que estes serviços são implementados, os *datacenters*.

Em termos gerais, há duas formas de implementar um mecanismo de controle de tráfego para as máquinas virtuais de um *datacenter*: a primeira é delegar o controle ao cerne da rede, modificando os componentes de rede para que eles controlem as taxas de transferência de dados para cada servidor. Esta forma de controle permite a implementação de compromissos entre taxas de transmissão e recepção que tratariam o conflito descrito no parágrafo anterior, mas tem a desvantagem de implicar em uma implementação complexa, de difícil manutenção e com custo potencialmente proibitivo. A segunda alternativa é implementar o controle nas pontas da rede, delegando o controle aos servidores, que utilizariam mecanismos de controle de tráfego em *software* para gerenciar o tráfego das máquinas virtuais. As vantagens desta abordagem são sua independência dos componentes de rede do *datacenter* e a sua flexibilidade comparável a dos mecanismos de controle que atuam no cerne da rede.

Apesar de que, à primeira vista, as vantagens de se implementar o controle de tráfego nos servidores sejam significativas, esta forma de controle possui uma limitação

fundamental: um servidor pode controlar de forma explícita apenas o tráfego que ele origina, mas não o que recebe. A natureza desta limitação vem do fato que um servidor, quando receptor de uma transmissão, só pode decidir como tratar os dados depois que eles atravessam seu enlace. Nesse ponto, caso o enlace do servidor esteja próximo a um nível de sobrecarga, o receptor pode enviar uma mensagem de controle ou descartar os dados que recebe, esperando que o transmissor detecte e reaja a essas situações. Este método é efetivo para fluxos de rede governados por protocolos de comunicação com controle de fluxo, como TCP, mas não para fluxos que não possuem tais mecanismos, como UDP. Quando os transmissores ignoram as medidas do servidor e mantêm suas taxas de transmissão, o enlace de entrada do servidor pode sofrer congestionamento, prejudicando o desempenho das máquinas virtuais que ele executa e desperdiçando recursos de rede dos transmissores, já que o servidor vai descartar parte dos dados que recebe.

Assim, o desafio de implementar um mecanismo de controle de tráfego, que controle de forma confiável tanto o tráfego de transmissão quanto de recepção das máquinas de um *datacenter*, consiste em elaborar uma solução que seja independente dos componentes da malha de rede e que contorne a limitação natural dos mecanismos de controle de tráfego de recepção dos servidores. Para tal, este trabalho sugere utilizar o nível adicional de controle provido pela camada de virtualização dos servidores para expor novas formas de controle de tráfego, que suprimam as limitações das tecnologias atuais.

1.2 Objetivos

O objetivo deste trabalho é, utilizando as capacidades de controle adicionais oferecidas pela camada de virtualização, construir um mecanismo de controle de tráfego executado pelos servidores e transparente às máquinas virtuais, que supere as limitações dos mecanismos de controle atuais, oferecendo o controle confiável de taxas de transmissão e recepção de cada máquina virtual, independente das características do fluxo de rede gerado pelas aplicações dos clientes.

A construção desta solução também demanda a definição de um modelo de rede, que seja claro para os clientes e que trate os conflitos gerados ao oferecer garantias de transmissão e recepção. Como discutido anteriormente, estes conflitos ocorrem em casos em que são feitas múltiplas transmissões, originadas de máquinas virtuais diferentes, para uma mesma máquina virtual de destino. Nestes casos, a soma das taxas garantidas de transmissão das origens de dados pode exceder a taxa garantida de recepção do destino, o que impossibilita garantir os limites de transmissão das origens

sem violar o limite de recepção do destino.

1.3 Contribuições

A principal contribuição deste trabalho é a criação do *Gatekeeper*, um mecanismo distribuído de controle de tráfego para os servidores de um *datacenter*, que se distingue dos mecanismos atuais ao permitir não apenas o controle da transmissão de dados, como também controlar de forma confiável o tráfego de recepção para cada máquina virtual, independente do comportamento de rede das máquinas virtuais. O *Gatekeeper* estabelece um modelo de controle de tráfego no qual o tráfego das máquinas virtuais é classificado em conexões através de um mesmo *switch* virtual, com limites de transmissão e recepção individuais. Esse modelo permite implementar políticas de controle que solucionam conflitos entre as garantias de transmissão e recepção, sem torná-lo complexo demais para os clientes que executam máquinas virtuais no *datacenter*.

O *Gatekeeper* é transparente para as máquinas virtuais executadas nos servidores, oferecendo aos usuários a impressão de possuírem enlaces com capacidades garantidas diferentes das capacidades reais dos *hardware* dos servidores. Além de garantir taxas mínimas, o mecanismo aproveita os recursos ociosos do servidor quando, por exemplo, uma das máquinas virtuais não usa sua alocação mínima de transferência, partilhando a capacidade de enlace ociosa entre as máquinas virtuais que apresentam demanda, permitindo que a máquina virtual ociosa reaproprie seus recursos quando necessário. Isto significa que o *Gatekeeper* implementa um escalonamento de recursos de rede que *conserva trabalho*, ou seja, sempre que há demanda por recursos de rede, o mecanismo usa toda a capacidade disponível para atender esta demanda, ao contrário dos mecanismos atuais, que por limitações da tecnologia de controle ou por questões de confiabilidade, não permitem que uma máquina virtual ultrapasse sua alocação de recursos, mesmo quando há uma quantidade maior de recursos ociosos no servidor.

Embora a implementação do *Gatekeeper* avaliada tenha sido desenvolvida como um programa executado no nível de usuário de um sistema *Linux*, a arquitetura do *Gatekeeper* é flexível, permitindo sua implementação em camadas mais próximas ao *hardware*, como no próprio *kernel* do *hypervisor* Xen ou até mesmo no *firmware* de interfaces de rede modernas.

1.4 Estrutura

O restante desta dissertação é dividida desta forma: O Capítulo 2 enumera os conceitos relacionados, como as tecnologias de virtualização empregadas e o modelo de controle de tráfego do Linux. A proposta do *Gatekeeper*, sua arquitetura, detalhes de implementação e as medições feitas para configurá-lo são apresentados no Capítulo 3, assim como uma comparação com os trabalhos relacionados. O Capítulo 4 discorre sobre as avaliações conduzidas para testar a corretude do *Gatekeeper* e o seu impacto no desempenho de aplicações reais.

Capítulo 2

Conceitos relacionados

Antes de detalhar a proposta do *Gatekeeper* e sua implementação, é pertinente visitar alguns conceitos básicos relativos ao ambiente para o qual ele foi projetado e as tecnologias utilizadas em sua implementação. Neste capítulo, será feita uma breve introdução à virtualização, discorrendo sobre seus conceitos, aplicações e diferentes abordagens. Em seguida, será detalhado o ambiente de virtualização escolhido para a implementação do protótipo do *Gatekeeper* utilizado na avaliação, o ambiente *Xen*, incluindo os seus mecanismos de isolamento de recursos de rede. Este capítulo é encerrado com uma descrição do mecanismo de controle de tráfego em *software*, oferecido pelo *kernel Linux*, utilizado como base na implementação do *Gatekeeper*.

2.1 Virtualização

O conceito de virtualização não é recente, tendo sido criado pela IBM na década de 60 para prover acesso concorrente aos seus *mainframes*. Neste modelo, são criadas máquina virtual que representam instâncias de uma máquinas físicas, com cada usuário tendo a ilusão de acessar uma máquina real, isolada e protegida das outras máquinas virtuais sendo executadas no mesmo *mainframe*. Na década de 90, a utilização de máquinas virtuais aumentou devido ao surgimento de diversos sistemas operacionais e opções de *hardware*, pois elas permitiam a execução de sistemas operacionais em *hardware* que outrora não os suportaria.

O termo *virtualização* é definido por Nanda em [38] como a tecnologia que combina ou divide recursos computacionais para executar um ou mais sistemas operacionais. Em um ambiente virtualizado, uma entidade é responsável pela criação e gerência das máquinas virtuais que executam independentes e isoladas umas das outras em um mesmo servidor. Esta entidade costuma ser chamada de Monitor de Máquinas

Virtuais, ou MMV. As aplicações das propriedades de isolamento e abstração das tecnologias de virtualização são inúmeras, tendo como exemplos:

- **Consolidação de servidores**: em ambientes como *datacenters*, que possuem diversos servidores executando serviços diferentes, é possível que alguns servidores sejam subutilizados quando a carga total de seus serviços seja menor do que a sua capacidade. Com o uso de um ambiente virtualizado, um servidor pode executar múltiplos serviços simultaneamente, mantendo o isolamento e utilizando seus recursos de maneira mais eficiente;
- **Consolidação de aplicações**: Aplicações antigas podem requerer *hardware* específico, potencialmente indisponível. A camada de virtualização pode simular esse *hardware*, satisfazendo os requisitos de execução da aplicação mesmo quando o *hardware* real necessário não é disponível;
- ***Sandboxing***: O uso de máquinas virtuais permite a execução de aplicações não-confiáveis em ambientes seguros e isolados, provendo um ambiente de execução seguro;
- **Suporte a múltiplos ambientes de execução**: Quando um sistema precisa executar múltiplas aplicações, com dependências de *hardware* e *software* distintas e conflitantes, a tecnologia de virtualização permite que sejam criadas múltiplas máquinas virtuais, com ambientes de execução diferentes, para oferecer ambientes de execução distintos e isolados para essas aplicações;
- **Depuração e teste de aplicações**: Algumas aplicações podem apresentar erros que são difíceis de serem depurados, por ocorrerem apenas em plataformas de *hardware* ou ambientes de execução específicos. Em um ambiente virtualizado, o desenvolvedor pode criar máquinas virtuais com as configurações específicas que supostamente geram os erros, criando testes determinísticos e reprodutíveis, além de possuir acesso a detalhes da execução de *software* de baixo nível, como sistemas operacionais e *drivers* de dispositivos, que podem auxiliá-lo na depuração.

As diferentes tecnologias de virtualização diferem primariamente no nível de abstração apresentado. Usando a categorização definida por [38], a virtualização pode ocorrer no nível de conjunto de instruções, emulando as instruções do processador de uma arquitetura para outra [22, 54]; no nível da camada de abstração de *hardware* (*hardware abstraction layer*, ou HAL) , com o MMV se colocando entre o *hardware* e o sistema operacional [5, 55]; no nível do sistema operacional, em que o MMV controla

o acesso às chamadas de sistema, executando outras aplicações em ambientes isolados [29, 30, 12]; no nível de bibliotecas de nível de usuário, no qual os desenvolvedores usam bibliotecas específicas para acessar os recursos virtualizados e no nível de aplicação, no qual o MMV é executado com uma aplicação que cria uma máquina virtual, como as máquina virtuais Java [32].

Embora o *Gatekeeper* possua uma arquitetura adaptável, seu foco são soluções de virtualização no nível da camada de abstração de *hardware*. Assim, as próximas seções discutirão os conceitos deste método de virtualização e do ambiente de virtualização escolhido para a implementação avaliada: o MMV Xen.

2.1.1 Virtualização em nível de camada de abstração de hardware (HAL)

A principal característica da virtualização em nível de HAL é o sacrifício da compatibilidade entre arquiteturas de processadores diferentes em troca de uma solução especializada e eficiente. Neste tipo de virtualização, a arquitetura dos processadores da máquina física e das máquinas virtuais deve ser a mesma, o que apesar de diminuir a abrangência da solução, aumenta a eficiência da camada de virtualização, pois o MMV não precisa mais interpretar as instruções do processador oriundas das máquinas virtuais para a processador da máquina física.

O princípio desta tecnologia é que as instruções privilegiadas executadas nas máquinas virtuais, que não têm permissão para tal, causam *traps*, que são capturados pelo MMV que, por sua vez, as inspeciona e executa as instruções no processador. Esse controle garante o isolamento entre máquinas virtuais. Entretanto, nem todas as instruções privilegiadas executadas pelas máquinas virtuais podem ser capturadas, pois ao serem executadas, elas falham silenciosamente ao invés de causarem *traps*. Os meios encontrados para contornar esse problema incluem métodos de análise e modificação dinâmica de código e mudanças nos próprios sistemas operacionais das máquinas virtuais. As diferenças destes métodos dividem as soluções de virtualização em nível de HAL em duas categorias:

- **Virtualização completa:** em ambientes de virtualização completa, as máquinas virtuais executam sistemas operacionais sem quaisquer modificações. Com isso, capturar todas as instruções privilegiadas executadas pelas máquinas virtuais requer a aplicação de métodos complexos de análise dinâmica de código, como a tecnologia usada pelo VMware para reescrever partes do código que devem ser repassadas para o MMV em tempo de execução. Isto impõe um custo adicional na

execução das máquinas virtuais, que embora possa ser atenuado usando artifícios como *caching* para as porções de código mais utilizadas, ainda é significante.

- **Paravirtualização:** o ponto principal da técnica de paravirtualização é a introdução de mudanças localizadas nos sistemas operacionais executados pelas máquinas virtuais, expondo pontos de acesso para dispositivos virtuais, com o objetivo de simplificar a camada de virtualização e alcançar melhor desempenho e escalabilidade. O Denali, proposto por Whitaker em [55], provê acesso a instruções, registradores e dispositivos de E/S virtuais para os sistemas operacionais, expondo instruções simplificadas e criando instruções otimizadas para as instruções mais simplificadas. Estas decisões arquiteturais diminuem o custo inserido pela camada de virtualização e faz com que as máquinas virtuais utilizem menos recursos.

  Os criadores do Xen [5], por sua vez, argumentam que a solução proposta pelo Denali é específica para máquinas virtuais simples, que executam apenas serviços de rede, também apontando que é possível tirar melhor proveito das oportunidades atreladas à paravirtualização. Tais oportunidades incluem permitir que as máquinas virtuais diferenciem recursos reais de virtuais, efetivamente quebrando a ilusão de execução em um sistema real, mas oferecendo novas ferramentas para maximizar o desempenho de aplicações sendo executadas em ambientes virtuais. Na próxima seção, serão discutidos os conceitos básicos do Xen, e suas características importantes na implementação deste trabalho.

2.1.2 Xen

Embora virtualização completa possua o benefício de permitir a execução de sistemas operacionais sem modificações, esta vantagem traz um custo adicional, dado que a arquitetura mais popular, a *x86* não foi projetada para virtualização[1]. O exemplo mais crítico das dificuldades enfrentadas ao virtualizar esta arquitetura é que algumas instruções privilegiadas, quando executadas por um agente sem os privilégios necessários, falham silenciosamente ao invés de causarem *traps*. Como tais *traps* seriam a única forma do MMV capturar as chamadas a essas instruções, é necessário o uso de técnicas de inspeção e modificação dinâmica de código para inserir chamadas ao MMV em trechos de código que executam instruções privilegiadas.

[1]Embora a popularização do modelo de virtualização tenha dado origem a mecanismos de suporte em *hardware* à virtualização, eles não mitigam o problema causado por instruções privilegiadas que falham silenciosamente.

A *paravirtualização* surgiu como uma alternativa ao modelo de virtualização completa, tendo como característica a modificação dos sistemas operacionais das máquinas virtuais para adaptá-los aos ambientes virtualizados em que são executados. Esse modelo tem como principais proponentes o Denali [55] e o Xen [5]. Enquanto o Denali apenas suporta máquinas virtuais leves, voltadas para aplicações específicas, o Xen permite a execução de sistemas operacionais completos, mantendo a interface de acesso ao sistema e adicionando funcionalidades como *timers* virtuais, úteis para tarefas sensíveis ao tempo como o tratamento de *timeouts* TCP[2], e dispositivos virtuais, que definem um ponto de acesso unificado aos dispositivos físicos de um sistema e que são gerenciados por uma máquina virtual especial, denominada *Driver domain*. A Figura 2.1 mostra a arquitetura de um ambiente Xen, com seu MMV controlando o acesso ao *hardware* e executando três máquinas virtuais: *VM A*, *VM B* e o *Driver domain*.

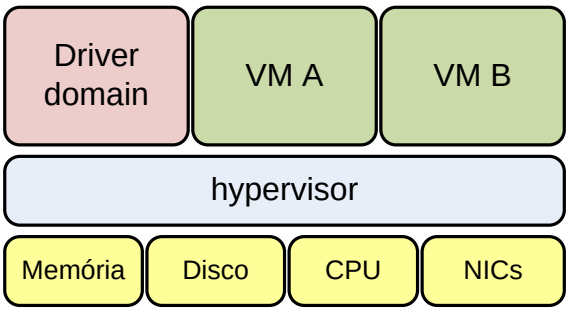


Figura 2.1. Arquitetura de um ambiente Xen executando três máquinas virtuais, incluindo do *Driver domain*.

Além da exposição de conceitos de alto nível como *timers* e dispositivos virtuais para as aplicações de um sistema operacional virtualizado, as principais características do Xen são:

- **Gerência simplificada de memória:** Nas soluções de virtualização completa e no Denali, o MMV trata todos os acessos à memória das máquinas virtuais, o que é computacionalmente caro. Em um ambiente Xen, quando uma máquina virtual tenta acessar uma porção de memória alocada para ela, o MMV Xen é invocado apenas para autorizar o acesso. Caso o acesso seja permitido, a máquina virtual pode acessar diretamente a porção de memória, sem intervenção do MMV. Acessos subsequentes para a mesma porção de memória não necessitam de autorização, e pedidos de autorização para múltiplas porções de memória podem ser agregados em *batches*;

[2]Transmission Control Protocol

- **Gerência de CPU:** Para proteger o próprio MMV de máquinas virtuais que executam sistemas operacionais maliciosos, o Xen executa os sistemas operacionais das máquinas virtuais em um nível de privilégio mais baixo do que o MMV. Usando um princípio semelhante ao da gerência de memória, as máquinas virtuais podem registrar tratadores para as chamadas de sistema dos sistemas operacionais, que após autorizados, não necessitam ser intermediados pelo MMV. Para multiplexar interrupções entre as diferentes máquinas virtuais, o Xen usa um sistema de eventos, em que as máquinas virtuais se registram em uma lista de observadores para separados tipos de interrupções, sendo notificadas pelo o MMV quando novas interrupções deste tipo são geradas;

- **Dispositivos de E/S virtuais:** Ao invés de apresentar versões virtualizadas do hardware físico para as máquinas virtuais, o Xen utiliza dispositivos virtuais, que possuem a mesma interface, independente do *hardware* que acessam. Além desta separação entre *hardware* físico e virtual, é inserida uma máquina virtual especializada para controlar o acesso aos dispositivos de E/S, chamada *Driver domain*, ou *dom0*. O *Driver Domain* é o único que utiliza os *drivers* reais para acessar diretamente os dispositivos de E/S reais. A vantagem primordial deste método é que, mesmo que o *hardware* físico dos dispositivos de E/S seja completamente modificado, apenas o *Driver Domain* deverá ser adaptado, pois enquanto não houver mudanças na interface de acesso aos dispositivos virtuais, os *drivers* das máquinas virtuais não precisam ser modificados.

Estes conceitos fazem com que máquinas virtuais Xen atinjam níveis de desempenho agregado semelhantes aos de máquinas reais [9, 5, 44]. Não obstante, a adição de uma máquina virtual específica para o acesso aos dispositivos de E/S permite a implantação de políticas de acesso de alto nível a esses dispositivos, o que provou ser essencial na implementação do *Gatekeeper*. O funcionamento dos dispositivos de E/S virtuais serão discutidos na próxima seção.

2.1.2.1 Dispositivos de E/S virtuais

Ao invés de criar apenas versões virtualizadas para *drivers* de dispositivos de E/S, os desenvolvedores do Xen decidiram desacoplar o *hardware* virtual do *hardware* real, introduzindo uma máquina virtual especial, chamada *Driver Domain*, que controla o acesso aos dispositivos de E/S físicos e exporta interfaces virtuais para as outras máquinas virtuais. A vantagem desta abordagem é que apenas o *driver* do sistema

operacional do *Driver domain* precisa suportar o dispositivo, para disponibilizar o dispositivo para todos os possíveis sistemas operacionais das outras máquinas virtuais.

O nome dado para esse modelo é *Split driver model*, ou *modelo de driver particionado*. Antes de detalhá-lo, é pertinente introduzir parte da terminologia usada pelo Xen: com exceção do *Driver domain*, as máquinas virtuais em um ambiente Xen são chamadas *guests*; O MMV é chamado *hypervisor*; os *drivers* do *Driver domain* são chamados *backend drivers*, enquanto que os drivers virtualizados dos *guests*, *frontend drivers*.

A Figura 2.2 mostra a arquitetura Xen para controle de dispositivos de E/S. Para fornecer acesso a um dispositivo de *hardware*, o Xen executa um par de *drivers* para cada dispositivo de E/S virtual: um no *Driver domain*, chamado *backend driver* e outro no *guest*, chamado *frontend driver*. Quando sistema operacional de um *guest* precisa acessar um dispositivo de *hardware*, ele se comunica com o *frontend driver*, que estabelece um canal de comunicação com o respectivo *backend driver* do *driver domain*. O *backend driver*, por sua vez intermedia o acesso dos *guests* ao dispositivo de *hardware*.

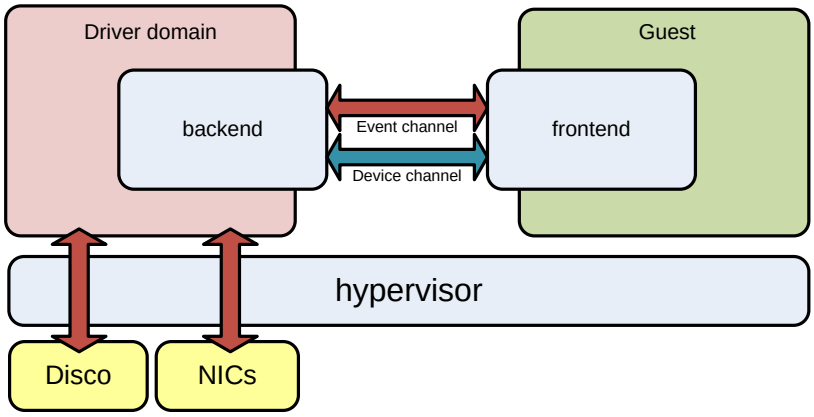


Figura 2.2. Arquitetura do *split driver model*. O *backend* é o único que acessa os *drivers* do dispositivo real.

O Xen não repassa as interrupções de *hardware* geradas pelos dispositivos para as máquinas virtuais. O *hypervisor* captura as interrupções, e após uma validação ele as repassa para a máquina virtual de destino, na forma de eventos. Para isso, é criado um canal de comunicação entre o *backend* e *frontend driver*, o *event channel*. Para a transferência de dados, é criado um canal de transferência assíncrono, chamado *device channel*. Ambos canais são ilustrados na Figura 2.2.

Para ilustrar o modelo de *driver* particionado proposto pelo Xen, vamos usar como exemplo a forma com a qual os *guests* Xen acessam as interfaces de rede físicas da máquina física. Quando uma interface de rede virtual deve ser inicializada em

um *guest*, é criado um par de *drivers*: um chamado *netback*, no *Driver Domain*, e um *netfront*, no *guest*. O *Driver Domain* é responsável por estabelecer a conexão inicial entre o seu *netback* e o *netfront* de cada dispositivo virtual, através de canais de comunicação chamados *event channel* e *device channel*.

Os sistemas operacionais do *Driver Domain* e dos *guests* acessam os *drivers netback/netfront* como interfaces de rede comuns. A Figura 2.3 ilustra quais interfaces de rede os sistemas operacionais *Linux* observam, para um sistema com uma interface de rede física, executando um *guest* de identificador *U* com duas interfaces de rede virtuais. Note que o *Driver Domain* detecta a interface física como *eth0* e os *drivers netback* do *guest* como *vifU.0* e *vifU.1*, enquanto o *guest* detecta duas interfaces de rede, uma para cada *netfront*, *eth0* e *eth1*.

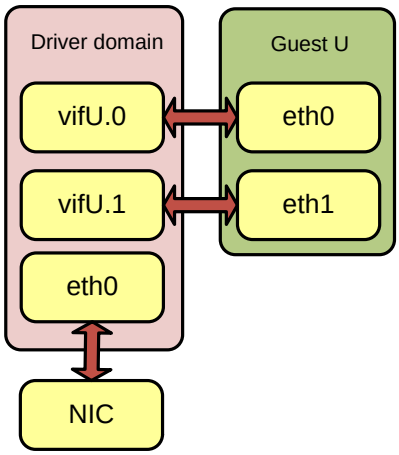


Figura 2.3. Interfaces de rede detectadas pelos sistemas operacionais do *Driver domain* e de um *guest* U, quando há apenas uma interface de rede física e o guest U possui duas interfaces virtuais.

* Entretanto, apenas inicializar as interfaces de rede no *Driver domain* e nos *guests* não é suficiente para que os *guests* possam acessar o dispositivo de E/S físico. É preciso conectar, no *Driver domain*, as interfaces físicas e virtuais, fazendo com que ele atue como um comutador ou roteador de pacotes. Na seção seguinte, será discutida a configuração mais utilizada, na qual o *Driver Domain* faz a conexão entre os *guests* através de uma ponte virtual.

2.1.2.2 Configuração de rede no Xen

Como mencionado na seção anterior, apenas criar os pares *netback/netfront* e as interfaces de rede respectivas no *Driver domain* e nos *guests* não é o suficiente para garantir aos *guests* acesso às interfaces de rede físicas. Para cada dispositivo virtual de um *guest*, é criada uma interface de rede no *Driver domain*, correspondente ao *netback*

do dispositivo virtual. Para permitir que o *guest* acesse a interface física, é necessário conectar esta interface à interface virtual do *guest*. Há duas formas comuns de fazer esta conexão: na primeira, e mais utilizada, o *Driver Domain* cria uma ponte de que liga as interfaces virtuais à interface física; na segunda, o *Driver domain* configura a sua tabela de roteamento para redirecionar corretamente os pacotes que são enviados ou recebidos. Como a primeira forma é utilizada na implementação do *Gatekeeper*, também sendo a mais comum, ela será o foco desta seção.

Na configuração de rede em modo ponte, é criada uma ponte virtual para conectar as interfaces virtuais dos *guests* à interface física, com acesso controlado pelo *Driver Domain*. Neste modo, após a inicialização do sistema o *Driver Domain* cria uma ponte virtual em *software*, que conecta-se às interfaces dos *guests* e à interface física. Quando um pacote de dados é recebido pela interface física, a ponto usa o endereço MAC do seu destino para decidir qual *guest* deve receber o pacote. Uma característica importante dessa configuração é que ela permite o uso de um serviço DHCP, que configura automaticamente dos endereços IP dos *guests*, o que não é possível para outras configurações.

Tomando como exemplo um ambiente Xen, com o *Driver domain* tendo acesso a uma interface de rede física *eth0* e com mais dois *guests*, *Ga* e *Gb*, com interfaces virtuais *vifA* e *vifB*, respectivamente, as configurações resultantes seriam: O *Driver domain* renomeia sua interface física para *peth0* e cria uma ponte chamada *eth0*. Para conectar os *guests* à interface física (agora *peth0*), o *Driver domain* apenas conecta as interfaces *peth0*, *vifA* e *vifB* à ponte *eth0*. A Figura 2.4 mostra a configuração criada para o ambiente de exemplo.

Até este ponto, foram apresentados os conceitos básicos de virtualização e do Xen, um ambiente de virtualização em nível de *hardware abstraction layer*, que utiliza técnicas de paravirtualização para otimizar o uso da máquina física. Também foi discutido o modelo de *driver* particionado proposto pelo Xen para desacoplar detalhes de acesso ao *hardware* dos *drivers* de dispositivos de E/S das máquinas virtuais, usando como exemplo os dispositivos de rede. Este capítulo continua com uma discussão sobre conceitos gerais de controle de tráfego e sobre como o sistema operacional *Linux* pode controlar as taxas de transferência de suas interfaces de rede, explicando seus princípios, aplicações e limitações.

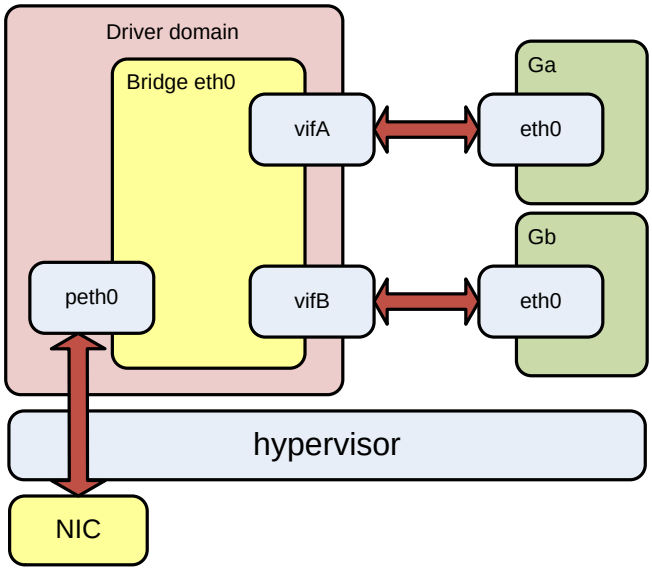


Figura 2.4. Configuração de rede do Xen para o modo ponte, para dois *guests* e uma interface física.

2.2 Controle de tráfego de rede

Antes de detalhar o funcionamento do mecanismo de controle de tráfego do *Linux*, é pertinente expor alguns conceitos básicos sobre o problema de controle de tráfego de rede em geral. Neste trabalho, o termo "controle de tráfego" se refere ao contexto de alocação de recursos de rede. O problema de alocação de recursos surge do fato de que, dado que os recursos são limitados, nem sempre é possível atender as demandas de todos os usuários, o que significa que os usuários receberão menos recursos do que demandam. Esta escolha da quantidade de recursos recebidos por cada usuário, quando a demanda excede a capacidade de serviço, configura o problema de alocação de recursos. Em uma rede, os recursos em questão são os enlaces dos servidores e os enlaces e filas dos elementos do cerne da rede, como roteadores e comutadores. Para fins de brevidade, o termo "roteador" será usado para se referir tanto a roteadores como comutadores, pelo restante desta seção.

Aplicar uma alocação de recursos para todos os usuários de uma rede é um problema não-trivial, pois os recursos estão distribuídos através de toda a rede. O problema é amplificado pela possibilidade de congestionamentos no cerne da rede, quando múltiplos usuários competem por um mesmo enlace em um roteador, sobrecarregando sua fila e provocando descartes de pacotes, ou nos enlaces de recepção dos servidores, quando os servidores recebem dados a uma taxa maior do que processam, o que também causa descartes. Assim, o problema de alocação de recursos de rede pode ser separado em duas partes complementares: escalonamento de recursos, que consiste em dividir cor-

retamente os recursos entre os usuários, e controle de congestionamento, que envolve o reajuste de taxas de transmissão para prevenir ou atenuar os efeitos de congestionamentos.

Ao definir as características de uma solução de alocação de recursos, é preciso definir as responsabilidades dos elementos do cerne da rede e dos servidores. Esta definição influencia diretamente os requisitos de escalonamento de recursos e controle de congestionamento. Ao, por exemplo, concentrar a solução no cerne da rede, fazendo com que os roteadores participem ativamente no escalonamento de recursos, diminui-se a possibilidade de congestionamentos, mas tal solução depende da precisão do escalonamento. Por outro lado, deixar o escalonamento em segundo plano, não exercendo qualquer controle sobre as taxas de transmissão dos usuários, intensifica a ocorrência de congestionamentos na rede, tornando necessário um mecanismo eficiente e ágil de controle de congestionamento. Na prática, no entanto, ambas abordagens são igualmente importantes, pois não há soluções definitivas para ambos problemas. A implementação de um escalonamento de recursos minimamente preciso é complexa, pois os recursos em questão estão distribuídos por toda a rede no passo que um controlador de congestionamento, por sua vez, não garante que não ocorrerão descartes antes que ele detecte e controle o congestionamento, o que desperdiça recursos de rede já escassos em situações de congestionamento.

Dada as diversas abordagens para o problema de alocação de recursos, é importante categorizar as principais alternativas e discutir alguns exemplos de implementações. Esta seção segue com uma categorização dos mecanismos de alocação de recursos de rede, seguida de uma descrição do escalonamento de filas dos roteadores de rede, continuando com uma discussão sobre mecanismos de controle de congestionamento do protocolo TCP.

2.2.1 Categorias de mecanismos de alocação de recursos

O problema da alocação de recurso de rede entre múltiplos usuários possui um diverso conjunto de abordagens, tornando complexa a tarefa de propor uma categorização unificada e abrangente para todas. Seguindo a proposta de Peterson em [43], podemos delinear conjuntos disjuntos de características, que podem ser combinadas na construção de uma solução de alocação. Estes conjuntos são:

- **Elemento atuador:** Mecanismos de alocação de recursos podem ser divididos em dois grandes grupos: aqueles nos quais a alocação é aplicada pelos elementos do cerne da rede, ou baseados no cerne, e aqueles baseados nos servidores, nos quais os servidores são responsáveis implementação da alocação. Esta divisão

não implica que os dois grupos são mutuamente excludentes, definindo apenas qual o elemento da rede com a maior responsabilidade.

Em mecanismos baseados no cerne da rede, cada roteador é responsável pelo redirecionamento de pacotes, decidir quando pacotes devem ser descartados e informar aos servidores quantos pacotes eles podem enviar. Em mecanismos baseados nos servidores, cada servidor observa as condições da rede (por exemplo: quantos pacotes estão efetivamente sendo entregues) e reajustam suas taxas de transmissão apropriadamente. Estas definições mostram que, mesmo quando a solução é baseada em um elemento da rede, a cooperação dos outros elementos ainda é necessária. No caso do mecanismo baseado no cerne, os roteadores esperam que os servidores sigam as taxas de envio que eles definem, enquanto que para os mecanismos baseados nos servidores, a entrega dos pacotes depende da lógica de roteamento do cerne da rede, por mais simples que seja;

- **Escalonamento de recursos:** Define como os servidores se comportam antes de usar os recursos de rede, dividindo os mecanismos em baseados em reserva e baseados em retorno. No primeiro caso, os servidores fazem uma requisição de recursos ao roteador antes de iniciar suas transmissões, só iniciando uma transmissão se o roteador decidir que o requisito não viola a alocação ou não o sobrecarrega. No caso de mecanismos baseados em retorno, os servidores iniciam suas transmissões sem quaisquer negociações com o roteador, ajustando suas taxas de transmissão de acordo com o comportamento da rede, como quando ocorrem perdas de pacotes;

- **Controle de transmissão:** Se refere à forma com a qual os servidores controlam suas taxas de transmissão. Neste caso, um mecanismo pode ser baseado em janelas, no qual os receptores informam aos transmissores a quantidade de dados que podem receber, ou baseado em taxas de transmissão, no qual os roteadores informam os servidores as taxas de transmissão que eles suportam ou permitem. Um exemplo de mecanismo baseado em janelas é o mecanismo de controle de fluxo do protocolo TCP, que será discutido na seção sobre seu controle de congestionamento.

Esta classificação abrange os tipos mais comuns de mecanismos de alocação recursos. Na próxima seção, será descrita a forma com a qual os roteadores gerenciam suas filas de pacotes, sua funcionalidade mínima em um mecanismo de alocação de recursos.

2.2.2 Gerência de filas

Seja qual for a complexidade do papel do roteador em um mecanismo de alocação de recursos, cada roteador deve implementar alguma forma de gerência para suas filas internas, que governe a forma com as quais os dados que passam por um roteador são armazenados antes de serem transmitidos. Tal forma de gerência é conhecida como *queueing discipline*, ou "política de enfileiramento". Uma política de enfileiramento tem as funções de alocar a banda e o espaço de armazenamento temporário do roteador, decidindo a ordem de envio dos pacotes na fila do roteador e quando pacotes devem ser descartados.

Existem diversas políticas de enfileiramento, que implementam algoritmos de escalonamento e critérios para descarte de pacotes distintos. Como exemplos de políticas de enfileiramento de roteadores, serão descritas as políticas *First In, First Out*, ou FIFO e a política *Fair Queueing*.

2.2.2.1 Política *First In, First Out* (FIFO)

Também conhecida como "*First Come, First Served*", esta é a política de enfileiramento mais simples. Seu escalonador transmite os pacotes na ordem em que o roteador os recebe, ou os descarta se a fila de armazenamento temporário estiver cheia no momento em que são recebidos. Sua simplicidade proporciona implementações rápidas e robustas, fazendo da FIFO a política mais utilizada nos roteadores da Internet [43].

2.2.2.2 Política *Fair Queueing*

A simplicidade da política FIFO traz consigo alguns pontos negativos. Por ela não diferencia as origens de tráfego, tratando todo tráfego igualmente, ela efetivamente delega a responsabilidade de controle de congestionamento e taxas de transmissão aos servidores da rede. Isso permite que, por exemplo, um servidor malicioso utilize uma grande fração dos recursos de redes ao gerar tráfego sem qualquer controle de congestionamento. No caso da Internet, por exemplo, nem todas as aplicações usam um protocolo de comunicação como o TCP, que possui controle de congestionamento. Para atacar este problema, foi proposta o algoritmo de *Fair Queueing*, ou FQ.

A ideia do FQ é separar o tráfego em fluxos, com base nos endereços de origem dos pacotes, mantendo filas separadas para cada um. Com o tráfego separado em filas, o roteador atende cada uma seguindo um algoritmo *round-robin*. Se a fila de algum fluxo é preenchida completamente, o roteador passa a descartar os pacotes que recebe deste fluxo, até que a fila esteja livre para receber novos pacotes. Como o FQ

não se comunica com os servidores para informar seu estado, ele também delega a responsabilidade de controle de congestionamento aos servidores.

As políticas de enfileiramento mostradas nesta seção, como já mencionado, consideram que os servidores da rede se responsabilizarão pelo controle de congestionamento. A próxima seção ilustra o mecanismo de controle de congestionamento baseado em servidores do protocolo predominante na Internet, o protocolo TCP.

2.2.3 Controle de congestionamento do protocolo TCP

Em seu princípio, o protocolo TCP não possuía um mecanismo de controle de congestionamento. Tal mecanismo foi introduzido logo após um período no qual Internet sofreu um colapso causado por congestionamentos. Nesse período, os servidores enviavam pacotes usando toda a capacidade de transmissão permitida por sua janela, eventualmente causando congestionamentos em algum roteador. Os transmissores, ao detectar que seus pacotes foram perdidos, os retransmitiam, deteriorando ainda mais a situação.

A ideia básica do TCP é permitir que o transmissor envie dados sem negociar com os roteadores, reagindo ao detectar eventos como perda de pacotes ou *timeouts*. Então, usando a classificação definida na Seção 2.2.1, o protocolo TCP é considerado como baseado em retorno. Resumidamente, a taxa de transmissão de um fluxo TCP é controlada indiretamente pelo receptor, que informa a quantidade de dados que ele pode receber e espera que os transmissores ajustem suas taxas de taxas de envio para acomodá-la. Os transmissores, por sua vez, aumentam suas taxas de envio linearmente e somente ao receber confirmações dos receptores. Adicionalmente, quando um transmissor detecta um *timeout* na recepção de seus pacotes, ele diminui pela metade sua taxa de transmissão, pois como *timeouts* são raros em situações que não configurem congestionamentos, o protocolo considera um *timeout* como um sinal de congestionamento. Este princípio, combinado com outras adições que aprimorar a eficiência do protocolo mas sem fugir da premissa inicial, tornam o TCP um protocolo robusto para a comunicação entre servidores de rede.

Este capítulo continua com a apresentação do mecanismo de controle de tráfego de rede oferecido pelo *kernel Linux*, que oferece aos servidores capacidades de controle de tráfego semelhantes as dos roteadores da rede, inclusive permitindo a utilização de políticas de enfileiramento.

2.3 Controle de tráfego de rede no *Linux*

Os *kernels Linux* recentes possuem funcionalidades que os tornam em versáteis ferramentas de controle de tráfego de rede. O elemento principal do modelo de controle de tráfego do *Linux* é a *queueing discipline*, ou "política de enfileiramento", que tem funcionalidades semelhantes as das políticas de enfileiramento dos roteadores de rede, mencionadas na seção anterior. Seu princípio é que, antes de serem enviados por uma interface de rede, os pacotes são enfileirados em filas específicas para cada interface de rede. Com os pacotes enfileirados, a política de enfileiramento escolhe quais pacotes devem ser enviados, descartados ou recolocados em cada fila. Esse conjunto de ações simples, combinado com um esquema de filtragem robusto e ferramentas em nível de usuário como o *iptables* e *ebtables* permitem controlar níveis de QoS[3], taxas de transferência, entre outras funções [2, 3, 26].

A Figura 2.5 mostra uma visão de alto-nível de como o *kernel Linux* trata os dados recebidos da rede ou que serão enviados. Quando um pacote de dados chega pela rede ou é enviado pelas camadas superiores da pilha de protocolos, o *kernel* decide se ele deve ser entregue ou redirecionado. Quando a escolha é o redirecionamento, o pacote é colocado na fila de saída, onde então atuam os mecanismos de controle de tráfego do Linux.

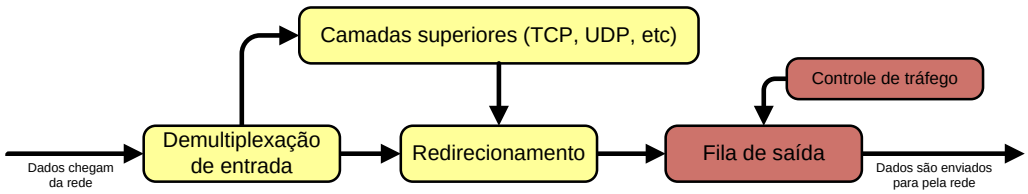


Figura 2.5. Tratamento de rede do *kernel Linux*. Dados recebidos da rede ou das camadas superiores são redirecionados ou repassados para a camada superior. O controle de tráfego atua no envio dos dados.

Cada interface de rede é associada a uma política de enfileiramento, que pode ser trocada ou modificada em tempo de execução. Existem várias opções de políticas, divididas em duas grandes categorias:

- **Políticas de enfileiramento *classful*:** Políticas que podem ser combinadas com elementos chamados *classes*, o pode lhes conferir funcionalidades ou comportamentos especiais. Algumas funcionalidades são exclusivas para políticas *classful*, como a formação de hierarquias de políticas e classes, que podem simular várias filas para um mesmo enlace, ou a aplicação de filtros de classificação;

[3] *Quality of Service*, ou Qualidade de Serviço. São níveis de desempenho, como taxa de transmissão ou latência, definidos pelo contrato entre o provedor e o cliente.

- **Políticas de enfileiramento *classless*:** Políticas mais simples, que tratam apenas uma fila. São normalmente utilizadas em conjunto com políticas *classful* na formação de hierarquias de políticas. Nessas hierarquias, políticas *classful* e classes definem características gerais dos membros da hierarquia e políticas *classless* definem como os dados serão escalonados para cada classificação possível.

Entre as políticas *classful* e *classless*, existem desde disciplinas simples, que tratam apenas uma fila usando um algoritmo *FIFO (first in, first out)*, até disciplinas complexas, que separam o tráfego em *classes* diferentes baseado em um ou mais *filtros*. A Figura 2.6 ilustra estes dois casos. No caso *a)*, a fila é única e apenas envia os pacotes na ordem em que eles são enfileirados; no caso *b)*, a política de enfileiramento é combinada com outras políticas e o tráfego que ela recebe é classificado por filtros. Cada política de enfileiramento contida pode tratar o tráfego de maneira diferente (inserindo latência, controlando a taxa de transmissão, etc).

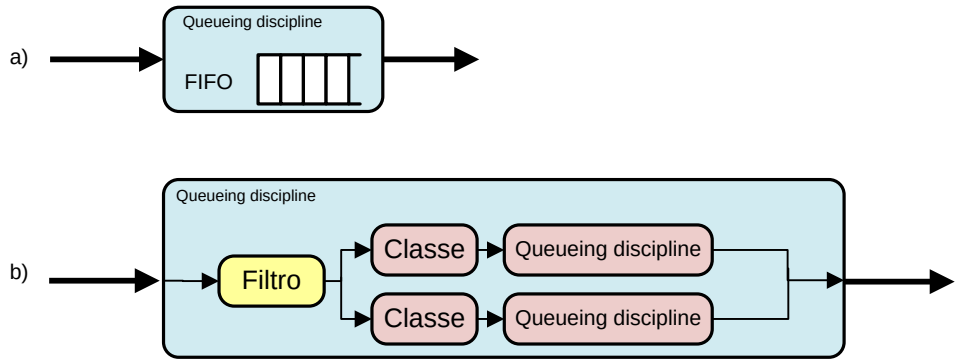


Figura 2.6. Duas combinações diferentes de políticas de enfileiramento para controlar a transmissão de dados de uma interface de rede qualquer.

Dentre as políticas de enfileiramento disponibilizadas pelo *kernel Linux*, duas foram escolhidas para a implementação do *Gatekeeper*: a política *classful Hierarchical Token Bucket* [13] e a política *classless FIFO* [26]. Como a configuração do controle de tráfego é pertinente à discussão sobre a implementação do *Gatekeeper*, é importante elaborar sobre as características de cada. Também é importante apontar as limitações do sistema de controle de tráfego, que fazem parte da motivação deste trabalho.

Dentre as diversas opções de políticas de enfileiramento oferecidas pelo *kernel Linux*, algumas requerem atenção especial, por serem fundamentais na implementação de controle de tráfego do *Gatekeeper*. Estas políticas *First in First Out (FIFO)* e a *Hierarchical Token Bucket (HTB)*, que serão descritas a seguir.

2.3.1 Política *First In, First Out* (FIFO)

Esta política de enfileiramento é a mais simples: o escalonador de uma política de enfileiramento *FIFO* retira dados da fila na ordem em que são enfileirados. Entretanto, a fila é finita. Assim, se o *kernel* tentar enfileirar dados quando a filar estiver cheia, os dados recebidos serão descartados. Há duas versões para esta política: *bfifo*, que usa *bytes* como medida de tamanho de fila, e *pfifo*, que usa pacotes. Para o *Gatekeeper*, foi criada uma versão especial da política *pfifo*, que registra o endereço IP do transmissor do último pacote descartado, dados necessários para a implementação do mecanismo de controle de congestionamento.

2.3.2 Política *Hierarchical Token Bucket* (HTB)

O *Hierarchical Token Bucket* é uma política *classful* que permite controlar a taxa de transmissão de um enlace, podendo também simular vários enlaces de menor capacidade usando o mesmo enlace físico. Esta simulação é feita ao separar tráfego em diferentes classes, que formam uma hierarquia, com configurações de taxa de transmissão e prioridades específicas. As regras para a formação de uma hierarquia HTB são:

1. A política de enfileiramento da interface de rede deve ser definida como uma política *HTB*. Esta política é chamada *raiz* e possui apenas um parâmetro, *rate*, que define a taxa máxima de transmissão da interface, que é obrigatório;

2. Uma ou mais classes HTB podem ser anexadas à política raiz. Estas possuem diversos parâmetros de configuração, sendo os principais *prio*, a prioridade da classe; *rate*, a taxa garantida de transmissão e *ceil*, a taxa máxima de transmissão. A diferença entre as taxas *rate* e *ceil* é que o escalonador deve garantir uma taxa de transmissão de no mínimo *rate*, ele pode alocar taxa de no máximo *ceil*, desde que outras classes não estejam utilizando suas alocações de enlace.

3. Classes pode ser anexadas a outras classes, formando uma árvore de classes;

4. A soma dos parâmetros *rate* das classes $\{c_1, c_2, ..., c_n\}$ anexadas a uma mesma classe c_p, não deve ser maior que o parâmetro *rate* de c_p;

5. Uma política de enfileiramento *classless* deve ser anexada a todas as classes que não possuem outras classes anexadas, ou seja, as folhas da árvore que representa a hierarquia de classes;

6. Como mencionado na seção de Visão Geral, são necessários filtros para classificar o tráfego. Estes filtros não devem ser anexados a uma política de enfileiramento específica, mas devem ser associados a um identificador especial. É uma solução menos intuitiva, mas que evita equívocos;

7. Opcionalmente, pode ser definida uma classe padrão, que recebe todo o tráfego não-classificado. Caso tal classe não seja definida, o tráfego que não for classificado é enviado na velocidade *máxima* permitida pelo *hardware*.

Estas regras permitem a formação de hierarquias complexas que, em conjunto com as capacidades de filtragem, oferecem um sistema de controle de tráfego versátil. Como exemplo, considere três clientes que usam o mesmo enlace para prover diferentes serviços, cada um gerando tráfego através de uma porta local diferente. A Tabela 2.1 lista os clientes, as portas usadas pelos serviços e as suas alocações de enlace.

Tabela 2.1. Lista de clientes, protocolos e portas de seus serviços e taxas de transmissão para o exemplo de configuração do HTB

Cliente	Protocolo	Porta	Taxa mínima	Taxa máxima
A	TCP	30051	100 Mbps	500 Mbps
B	TCP	30052	300 Mbps	900 Mbps
C	UDP	50010	500 Mbps	900 Mbps

Para configurar o enlace para aplicar o HTB, neste caso, é necessário configurar a hierarquia de classes e políticas, além dos filtros que classificarão o tráfego dos clientes. Os passos necessários para criar a hierarquia, ilustrada pela Figura 2.7 e configurar os filtros são:

1. Configurar a interface de rede para usar a política HTB, com $rate = 900\ Mbps$, que é a soma dos *rates* dos clientes A, B e C;

2. Adicionar a classe raiz HTB à política acima, também com com $rate = 900\ Mbps$;

3. Para cada cliente, adicionar uma classe à classe raiz, com $rate = taxa\ mínima$ e $ceil = taxa\ máxima$;

4. Adicionar as políticas de enfileiramento às classes folhas. Neste caso, foi escolhida a política *FIFO*;

5. Registrar os filtros para cada cliente. O mecanismo de controle de tráfego do *Linux* permite classificar o tráfego baseado, entre outras opções, no protocolo

de rede e na porta de origem e destino para protocolos TCP e UDP. Os filtros gerados para este exemplo podem ser vistos na Tabela 2.2

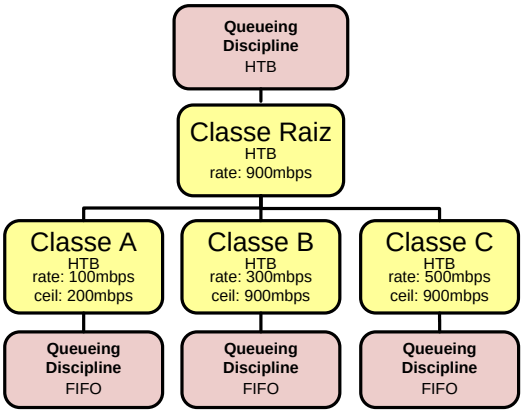


Figura 2.7. Hierarquia de classes e políticas de enfileiramento gerada para atender as configurações listadas na Tabela 2.1

Tabela 2.2. Filtros de classificação de tráfego gerados para o exemplo de configuração do HTB

Protocolo	Porta local	Classificar como?
TCP	30051	Classe A
TCP	30052	Classe B
UDP	50010	Classe C

2.3.3 Limitações

A princípio, as interfaces de rede de um sistema *Linux* possuem apenas uma política de enfileiramento, que controla apenas a transmissão de dados. Contudo, é possível *policiar* o tráfego de recepção, habilitando uma política chamada *ingress*. Esta política pode receber filtros como uma politica *classful*, mas não outras classes ou políticas de enfileiramento. A única maneira de controlar o tráfego, desta forma, é utilizando a capacidade dos filtros de aplicar um algoritmo de *token bucket* [51], que descarta seletivamente os pacotes recebidos.

Usando um módulo chamado *Intermediate Functional Block* (IFB), introduzido na versão 2.6.16 do *kernel Linux*, é possível aplicar as mesmas *queuing disciplines*, antes exclusivas para controle transmissão, ao controle de recepção de dados [27]. Em linhas gerais, o módulo IFB cria uma interface de rede em *software*, chamada *ifb0*. A única função desta interface é enviar todos os dados que ela recebe para o *kernel*, mas

como ela é uma interface de rede, uma política de enfileiramento pode ser utilizada para governar a transmissão (que neste caso, é da interface virtual *ifb0* para o kernel) dos dados. Desta forma, pode-se usar um filtro na política de enfileiramento *ingress* para repassar o tráfego para a interface *ifb0*, que por sua vez aplica sua política de transmissão, que a entrega para o *kernel*.

Por exemplo, para usar a política *FIFO* para controlar o tráfego de chegada em uma interface chamada *eth0*, a sequência de passos é:

1. Inicializar o módulo *ifb*, criando a interface *ifb0*;
2. Habilitar a política de enfileiramento *ingress* na interface *eth0*;
3. Adicionar um filtro à política *ingress* habilitada no passo anterior, que redireciona o tráfego para a interface *ifb0*;
4. Configurar a interface *ifb0* para usar a política de transmissão *FIFO*.

A configuração final é mostrada na Figura 2.8.

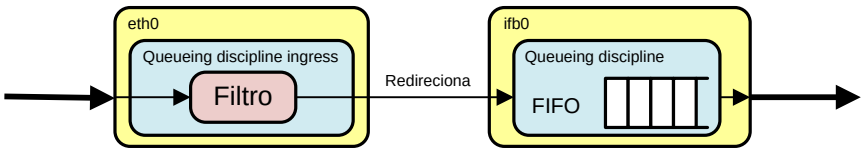


Figura 2.8. Usando o módulo *Intermediate Funcional Block* para aplicar políticas de enfileiramento de transmissão ao tráfego de recepção.

Todavia, mesmo ao se usar *Intermediate Functional Blocks*, não é possível contornar a limitação fundamental do controle de recepção: não há como o receptor controlar diretamente as taxas de envio dos transmissores. Fazendo um paralelo com o sistema postal, uma pessoa não tem controle sobre o volume de correspondência que recebe. Se o volume for muito alto, sua caixa de correio eventualmente ficará abarrotada. O problema é que os remetentes não são alertados sobre isso, ou simplesmente não consideram que a demora do destinatário em responder pode ser devida a sobrecarga. Assim, eles continuam a enviar o mesmo volume de correspondência.

Retornando ao caso do controle de tráfego de rede, o máximo que um receptor pode fazer para tentar controlar o tráfego de recepção, sem a participação de um agente externo, é descartar ou atrasar o recebimento de dados. Embora esta medida seja particularmente efetiva para controlar fluxos como TCP, que reagem a descartes e atrasos, o mesmo não se aplica para fluxos como UDP, que não possui um mecanismo de controle de fluxo como o TCP.

2.3.4 Considerações finais

Neste capítulo foram apresentados conceitos de virtualização e controle de tráfego, essenciais para a discussão da proposta do *Gatekeeper*. Em particular, vale salientar que as soluções de controle de tráfego abordadas até então possuem limitações que as tornam insatisfatórias para o problema de garantir taxas de transmissão e recepção para as máquinas virtuais de um *datacenter*. Especificamente, as soluções tradicionais de controle de tráfego, como as utilizadas para o tráfego da Internet e oferecidas pelo *kernel Linux*, são dependentes dos mecanismos de controle de fluxo dos protocolos de comunicação, o que as tornam inefetivas para tratar fluxos de protocolos que não possuem tal controle, como UDP.

O *Gatekeeper* almeja contornar esta limitação, ao usar a camada de virtualização para implementar um mecanismo distribuído de detecção de congestionamentos nos enlaces dos servidores e controle de fluxo. O próximo capítulo apresenta a solução proposta em detalhes.

Capítulo 3

Provendo garantias de tráfego com o *Gatekeeper*

Com o aumento da carga imposta aos *datacenters*, em parte devido à popularização do modelo de *cloud computing*, é essencial gerenciar os recursos dos servidores de forma eficiente. A utilização de ambientes de virtualização, como o Xen, compartilha eficientemente recursos como processamento e memória dos servidores, entre serviços executados por clientes diferentes. O mesmo não pode ser afirmado para os recursos de rede, pois as tecnologias de controle atuais ora se limitam a controlar tráfego que possui alguma forma de controle de fluxo, ora requerem o uso de dispositivos de *hardware* específicos na malha da rede.

O *Gatekeeper*, um mecanismo distribuído de controle de tráfego para ambientes virtualizados, foi criado com o objetivo de fornecer garantias de transmissão e recepção para as máquinas virtuais dos clientes de um *datacenter*. O mecanismo usa a camada de abstração, inserida pela camada de virtualização, para agir de forma transparente aos clientes e prover mecanismos de controle de congestionamento que permitem controlar tráfego sem controle de fluxo, como UDP, que não reage aos mecanismos de controle de tráfego atuais. Por último, a solução proposta usa os recursos de rede de forma eficiente, dividindo os recursos ociosos entre as máquinas virtuais. Os resultados da avaliação mostram que o *Gatekeeper* cumpre suas funções, partilhando as bandas de transmissão e recepção corretamente entre as máquinas virtuais dos servidores, mesmo com a presença de tráfego sem controle de fluxo.

Neste capítulo, serão descritos os objetivos do *Gatekeeper*, o modelo de rede adotado, os conceitos básicos do funcionamento do *Gatekeeper*, sua arquitetura e implementação no ambiente de virtualização Xen. Também serão discutidas as relações deste trabalho com outros trabalhos recentes na área de controle de tráfego, apontando

as contribuições do *Gatekeeper* e as diferenças de escopo e implementação.

3.1 Modelo de rede e objetivos

Antes de delinear os objetivos do mecanismo, é necessário definir o modelo de rede que é apresentado aos clientes do *datacenter*. Este modelo define o tipo e o escopo das garantias dadas para os clientes, fatores que influenciam diretamente o projeto do mecanismo e as métricas de avaliação. O modelo também precisa ser simples o suficiente para permitir que clientes que não são familiarizados com os conceitos de controle de tráfego possam fazer asserções e solicitações coerentes aos administradores do *datacenter*.

O modelo adotado pelo *Gatekeeper* é mostrado pela Figura 3.1. Neste modelo, o cliente tem a visão de que todas as máquinas virtuais do *datacenter* estão ligadas a um mesmo *switch*, através de um enlace com uma garantia mínima de banda BW. Este conceito se assemelha ao *hose model* [14, 19], modelo no qual o cliente especifica um conjunto de pontos de rede conectados, sendo que suas conexões possuem garantias de desempenho que devem ser asseguradas pelo provedor de serviços. A especificação de uma conexão é chamada *hose*, e inclui:

- A capacidade mínima de transmissão entre o ponto e a rede;
- A capacidade mínima de recepção entre o ponto e a rede;
- A capacidade agregada (transmissão e recepção) da conexão (*hose*).

A diferença entre o modelo adotado pelo *Gatekeeper* e o *hose model* está na definição das garantias de desempenho. Primeiramente, enquanto o *hose model* define parâmetros diferentes para transmissão, recepção e tráfego agregado, nosso modelo define apenas um parâmetro: a taxa garantida, que é aplicada para transmissão e recepção. Outro ponto é que, no modelo do *Gatekeeper*, o cliente pode alcançar uma taxa de transmissão ou recepção maior do que a mínima alocada, se os recursos alocados para outros clientes estiverem ociosos. No *hose model*, por sua vez, o cliente não alcança mais do que lhe foi alocado. Por último, é preciso estabelecer um compromisso entre as garantias de recepção e transmissão, pois oferecer garantias mínimas de transmissão e recepção simultaneamente são objetivos conflitantes. O modelo do *Gatekeeper* define que as garantias mínimas de recepção têm prioridade, sendo aplicadas mesmo quando isto implique em ferir as garantias de transmissão dos transmissores.

Com este modelo os objetivos do *Gatekeeper* são:

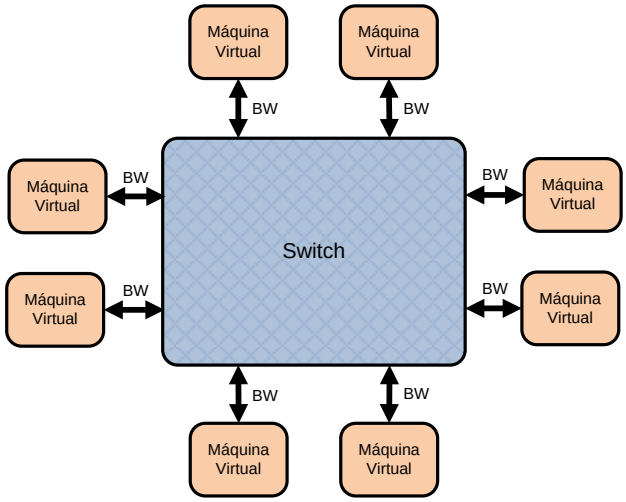


Figura 3.1. Modelo de rede adotado pelo *Gatekeeper*. As máquinas virtuais se conectam a um mesmo *switch* virtual, com garantias mínimas de transferência no enlace entre com elas o *switch*.

- Garantir uma taxa mínima de transmissão e recepção, para cada uma das máquinas virtuais dos clientes em um *datacenter*, priorizando as garantias de recepção em casos de múltiplas transmissões para um mesmo destino;
- Partilhar recursos ociosos entre os clientes, maximizando a utilização dos recursos de rede dos servidores.

Embora as ferramentas de controle de tráfego atuais sejam suficientes para atingir completamente o primeiro objetivo, elas falham em alcançar o segundo, pelos motivos discutidos no capítulo anterior: Embora os mecanismos de controle de tráfego de recepção sejam eficazes no controle de tráfego que possui controle de fluxo, como tráfego TCP, o seu princípio de funcionamento não se aplica para tráfegos que não possuem esse controle, como UDP. Isso se dá porque as únicas ações que o receptor pode tomar quando o seu enlace está congestionado são, ora descartar dados, ora atrasar seu recebimento. Enquanto o tráfego com controle de fluxo detecta e reage a descartes ou atrasos, tráfego sem esse controle não reage de qualquer forma.

A proposta do *Gatekeeper* é inserir um mecanismo na camada de virtualização dos servidores do *datacenter* que, além de controlar o tráfego de transmissão e recepção das máquinas virtuais, monitore o tráfego de recepção, detectando e mitigando congestionamentos. Delegar esta responsabilidade à camada de virtualização torna a solução transparente aos clientes, tornando a solução compatível com quaisquer sistemas operacionais suportados pelo ambiente de virtualização.

3.2 Trabalhos relacionados

Os mecanismos de controle de banda disponíveis nas interfaces de rede modernas, sistemas operacionais e monitores de máquinas virtuais apenas impõe limites máximos de envio para as máquinas virtuais, não provendo o controle confiável de recepção de dados provido pelo *Gatekeeper*. Além disso, estas soluções impõe limites fixos de envio, que implicam em desperdício de recursos quando uma ou mais máquinas virtuais deixam recursos ociosos.

Diversos esforços recentes propõe reformular as topologias de *Camada-2* utilizando equipamentos de baixo custo ou modificando as políticas de roteamento dos *datacenters* para prover alto tráfego de bisseção [1, 21, 20, 24, 37, 36, 47]. Estas novas tecnologias permitem que os servidores do *datacenter* possam usar a capacidade total de suas interfaces de rede, sem sobrecarregar a malha de rede. Assim, o *Gatekeeper* assume que o gargalo da rede deixa de ser a malha de rede, para ocorrer nos enlaces de rede dos servidores. Esta mudança justifica o foco do *Gatekeeper* em gerenciar os enlaces de acesso à rede dos servidores.

No âmbito dos sistemas de controle de taxas de transmissão, há diversas soluções baseadas em modificações nos roteadores que formam a malha da rede, que classificariam o tráfego gerado pelos clientes, controlando suas taxas de transmissão ao descartar [31] ou enfileirar [11, 41, 6, 17, 48] dados. Em [50], os autores argumentam que estes mecanismos não são suficientemente eficientes para tratar grandes volumes de dados, como os observados em roteadores de *backbone* de alta velocidade. Dado este requisito de eficiência, os autores propõe um mecanismo chamado *Core-Stateless Fair Queueing* (CFSQ).

Na política CFSQ, apenas os roteadores de fronteira da malha de rede mantém informações sobre os fluxos, enquanto o roteadores do núcleo da rede descartam dados de forma probabilística, usando marcações inseridas pelos roteadores de fronteira. O CFSQ permite uma alocação de banda próxima à justa, usando uma fração dos recursos requeridos por outras soluções. Todavia, o CFSQ assume que os servidores não são confiáveis, concentrando todo o controle no núcleo de rede. Em contraste, o *Gatekeeper* foi formulado para um ambiente de *datacenters* gerenciados, além de não requerer suporte em nível de *hardware* de rede, como roteadores ou pontes, pois ele concentra o controle de tráfego nos servidores.

O *Gatekeeper* também difere de outros mecanismos baseados em mudanças nos roteadores para a provisão de contratos de *QoS* e prevenção de ataques DoS[1] distribuídos [28]. Em particular, o *Gatekeeper* é complementar ao *Cloud Control* [46]. Enquanto

[1] *Denial of Service*

o *Cloud Control* propõe um mecanismo de controle de tráfego distribuído, usando um complexo protocolo distribuído para impor limitações de globais tráfego entre diferentes *datacenters*, o *Gatekeeper* foca em prover garantias de tráfego para um único *datacenter*, usando uma abordagem mais simples. Além disso, apesar de apresentar duas abordagens para controlar as taxas de transmissão, a sua abordagem mais eficiente não suporta tráfego que não possui controle de fluxo. O *Cloud Control* também requer mudanças nos elementos da malha da rede, requisitando funcionalidades mais complexas do que as oferecidas pelas pontes e roteadores usados pelas novas topologias de camada 2 de alto desempenho, que transferem o gargalo de rede da malha de rede para os enlaces dos servidores.

A crescente taxa de adoção do padrão *OpenFlow* [35] se deve à sua proposta de implementar pontes de rede programáveis, que podem prover roteamento diferenciado, entre outras funções, com base nos cabeçalhos dos pacotes que tratam. Uma série de projetos (SANE [34], Ethane [8], NOX [23]) utilizam o *OpenFlow* para implementar o gerenciamento centralizado do desempenho, segurança e propriedades gerais de rede. Este trabalho é largamente complementar ao *Gatekeeper*: fluxos individuais ou classes de tráfego usadas pelas máquinas virtuais poderiam ser gerenciadas de forma gerenciada, por sistemas como o *NOX*, enquanto o *Gatekeeper* proveria garantias de transmissão e recepção para as máquinas virtuais nos servidores individuais.

Concluindo, é pertinente esclarecer a diferença entre usar o *Gatekeeper* e utilizar apenas o mecanismo de controle de tráfego do *kernel Linux*, o TC. O TC oferece funcionalidades de controle de tráfego de transmissão e recepção, mas o seu algoritmo de controle de recepção é baseado em descartes de dados, o que é inefetivo para controlar tráfego que não possui controle de fluxo. Neste contexto, o *Gatekeeper* pode ser considerado complementar ao TC, detectando congestionamentos no enlace de entrada e reconfigurando os sistemas remotos através do TC, para controlar o tráfego de transmissão que causa o congestionamento.

3.3 Arquitetura

Os elementos principais da arquitetura de um sistema podem ser derivados de suas funcionalidades básicas. Sendo assim, dadas as funcionalidades básicas do *Gatekeeper*, que são: controle de tráfego de transmissão; controle de tráfego de recepção e controle de congestionamento, os principais elementos da arquitetura do *Gatekeeper* e suas respectivas funções são:

- **Escalonador de transmissão:** Divide o enlace de transmissão do servidor,

criando uma fila de transmissão para cada máquina virtual que ele executa. As máquinas virtuais são ligadas a um *switch* virtual, que classifica os fluxos de cada máquina virtual e os direciona a suas respectivas filas, com seu funcionamento representado pela Figura 3.2. Inicialmente, as filas são esvaziadas à uma taxa proporcional às taxas de transmissão das máquinas virtuais correspondentes, seguindo um algoritmo de *weighed fair queueing*. Quando uma ou mais máquinas virtuais não utilizam suas alocações de transmissão, os recursos ociosos são partilhados entre as outras máquinas virtuais no mesmo servidor;

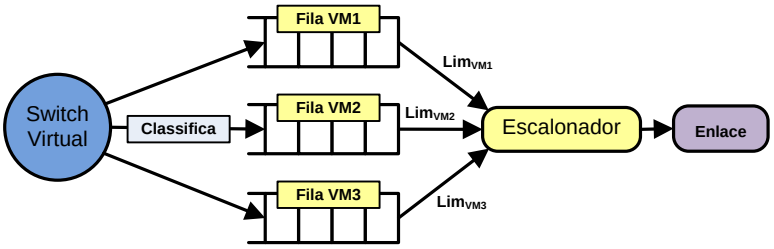


Figura 3.2. Escalonador de transmissão.

- **Escalonador de recepção:** Com função semelhante à do escalonador de transmissão, ele configura as filas de recepção para cada máquina virtual executada no servidor, classificando e redirecionando os dados que chegam da rede, de acordo com a Figura 3.3. A diferença é que, no caso da recepção, o escalonador não conhece a demanda dos transmissores, assim ele não sabe se, quando ele recebe dados de um transmissor à uma velocidade menor do que a alocada, é devido à uma caraterística real da aplicação do transmissor ou se é devido à um congestionamento na recepção.

  Para mover o gargalo de recepção do enlace do servidor para a camada de virtualização, o *Gatekeeper* configura o escalonador de recepção para ter um limite de recepção ligeiramente *menor* que o limite real. Isso faz com que sistema consiga detectar o princípio de um congestionamento antes que ele ocorra no enlace e tome as ações preventivas necessárias;

- **Controlador de congestionamento:** Sua função é, com base em dados coletados pelo escalonador de recepção, detectar e mitigar congestionamentos no enlace de entrada. Congestionamentos podem ser detectados observando as filas de recepção: quando o sistema recebe dados em uma taxa maior do que consegue processar, as filas tendem a ser completamente preenchidas; quando as filas de recepção estão cheias, o sistema então passa a descartas os dados que recebe. O

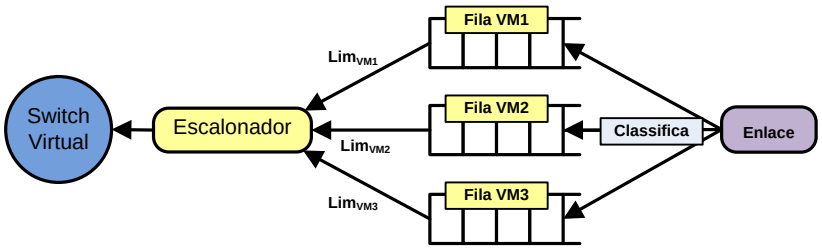


Figura 3.3. Escalonador de recepção.

controlador mede a quantidade de dados descartados, agindo quando a proporção de descartes excede um limiar previamente configurado. A razão para esperar que a quantidade de descartes atinja um limiar, ao invés de reagir imediatamente após o primeiro descarte, é que mesmo tráfego com controle de fluxo pode causar descartes antes de ajustar suas taxas de transmissão. Fluxos TCP, por exemplo, costumam causar descartes no início da conexão, durante a fase de *slow start* [49]. Para definir um valor para o limiar, foi conduzida uma caracterização do volume de dados descartados para tráfegos com e sem controle de fluxo. Esta análise é apresentada no capítulo de avaliação.

A detecção de um congestionamento significa que o sistema está recebendo tráfego de um ou mais transmissores que não estão reagindo aos descartes de dados. Neste caso, a função do *Gatekeeper* é escolher um destes transmissores e controlar sua taxa de transmissão, através da cooperação com o agente *Gatekeeper* executado no sistema que hospeda o transmissor. Para escolher o transmissor, o *Gatekeeper* observa as filas de recepção das máquinas virtuais: as filas que estiverem recebendo dados à uma taxa superior a suas alocações são provavelmente as filas afetadas pelo congestionamento. Considerando o conjunto Q_R de filas de recepção, o subconjunto C_R das filas potencialmente sobrecarregadas é dado pela Equação 3.1.

$$\begin{cases} C_R = \{q_{vm} \forall q_{vm} \in Q_R | \varphi_{vm} > 0\} \\ \\ \varphi_{vm} = banda\_usada_{vm} - banda\_alocada_{vm} \end{cases} \tag{3.1}$$

Dentre estas, a fila mais afetada é que possui o maior valor de φ_{vm}. Assim, é necessário controlar a taxa de envio das máquinas virtuais que enviam dados para a máquina virtual sobrecarregada. A escolha da máquina virtual transmissora a ser controlada segue uma estratégia gulosa: a máquina escolhida é a transmissora do último datagrama descartado pela fila mais sobrecarregada. Embora

uma alternativa mais segura seria analisar a fila sobrecarregada, escolhendo o transmissor que possui mais datagramas enfileirados, a melhor implementação possível para tal algoritmo teria complexidade *O(n)*, *n* sendo o tamanho da fila, em contraste com a complexidade *O(1)* do algoritmo seguindo a estratégia gulosa. Como este algoritmo deve ser executado pelo *kernel*, ele precisa executar o mais rápido possível, o que torna o algoritmo com complexidade *O(n)* uma alternativa inadequada e justifica o uso de uma estratégia gulosa.

Tendo escolhido o transmissor que deve ser controlado, o controlador de congestionamento envia uma mensagem de controle para o controlador executado pela máquina física que executa a máquina virtual transmissora. Quando o controlador de congestionamento, por sua vez, recebe uma mensagem de controle para uma de suas máquinas virtuais, ele reconfigura o escalonador de transmissão para diminuir a taxa de transmissão da máquina virtual reportada.

A Figura 3.4 ilustra os componentes e relações que formam a arquitetura do *Gatekeeper*. As máquinas virtuais usam o *switch* virtual para receber e enviar dados pela rede, enquanto o controlador de transmissão classifica e envia os dados que vem do *switch* virtual e o controlador de recepção classifica os dados que chegam da rede. O controlador de congestionamento monitora as filas de recepção e, quando a taxa de descartes na recepção passa de um limiar, envia uma mensagem de controle para um dos transmissores. Ao receber uma mensagem de controle, o controlador de congestionamento diminui a taxa de transmissão da máquina especificada na mensagem, para atenuar o congestionamento no receptor.

Na seção seguinte serão apresentados os detalhes da implementação do protótipo do *Gatekeeper*, detalhando as configurações dos escalonadores de recepção e transmissão e o funcionamento do controlador de congestionamento.

3.4 Implementação

O ambiente de virtualização Xen foi escolhido para a implementação do protótipo do *Gatekeeper*. O Xen foi escolhido por ser um projeto aberto, gratuito e principalmente por apresentar bom desempenho devido à sua tecnologia de paravirtualização. Como os detalhes da arquitetura do Xen e de suas configurações de rede foram examinados no Capítulo 2, o foco desta seção é expor como a arquitetura do *Gatekeeper* foi construída neste ambiente de virtualização.

Em um ambiente Xen o *Driver domain* é responsável pelo controle de acesso aos dispositivos de E/S, responsabilidade que se alinha com o objetivo do *Gatekeeper* que

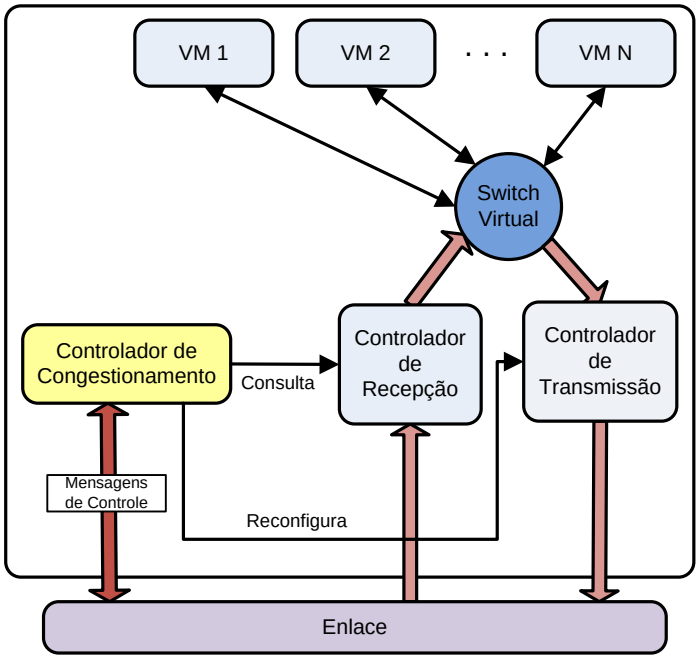


Figura 3.4. Arquitetura do *Gatekeeper*.

é, em essência, controlar o acesso ao enlace de rede. Desta forma, os componentes do *Gatekeeper* foram implementados como parte do *Driver domain* Xen. A arquitetura do *Gatekeeper* inclui a presença de um *switch* virtual, que pode ser representado pela ponte em *software* configurada pelo Xen no modo ponte (Figura 2.4). O controle de taxas de transferência, exercido pelos escalonadores transmissão e recepção, foi implementado através do uso de políticas de enfileiramento. Estas mesmas políticas fornecem as estatísticas usadas pelo controlador de congestionamento, que se comunica com os controladores de agentes *Gatekeeper* remotos através de mensagens UDPx. Uma visão geral da implementação da arquitetura do *Gatekeeper* em um ambiente Xen pode ser observada na Figura 3.5.

A configuração das políticas de enfileiramento de transmissão e recepção, como também a detecção e controle de congestionamentos são implementados por um agente, que é executado pelo *Driver domain* após sua inicialização. Este agente é composto por programas *Python*, *C* e *scripts Bash* que, respectivamente, controlam o fluxo de execução, coletam estatísticas das políticas e as configuram na inicialização do agente, ou em resposta à uma mensagem de controle de congestionamento. O fluxo de execução do agente é ilustrado pelo Algoritmo 1.

As próximas seções detalharão este fluxo, descrevendo as implementações das funções de controle de tráfego, detecção e controle de congestionamento.

```
1: # Lê a lista das máquinas virtuais locais do arquivo de configuração
2: vms = read_configuration()
3:
4: # Configura escalonadores de transmissão e recepção
5: initialize_transmit_scheduler(vms)
6: initialize_receive_scheduler(vms)
7:
8: loop
9:   # Coleta estatísticas dos escalonadores
10:  stats = collect_scheduler_statistics(vms)
11:
12:  # Executa algoritmo de detecção e controle de congestionamento
13:  if detect_congestion(vms, stats) then
14:    handle_congestion(vms, stats)
15:  end if
16:
17:  # Trata mensagens de controle de congestionamento recebidas
18:  control_messages = receive_congestion_control_messages()
19:
20:  # Se não há mensagens de controle, deve-se recuperar gradualmente as taxas
     de transmissão das máquinas virtuais locais
21:  if control_messages = [] then
22:    for all virtual_machine in vms do
23:      recover_transmission_rate(virtual_machine)
24:    end for
25:  else
26:    # Se há mensagens de controle, diminuir taxa de transmissão das máquinas
       virtuais escolhidas
27:    for all msg in control_messages do
28:      decrease_transmission_rate(msg.virtual_machine)
29:    end for
30:  end if
31: end loop
```

Algoritmo 1: Fluxo do agente *Gatekeeper*

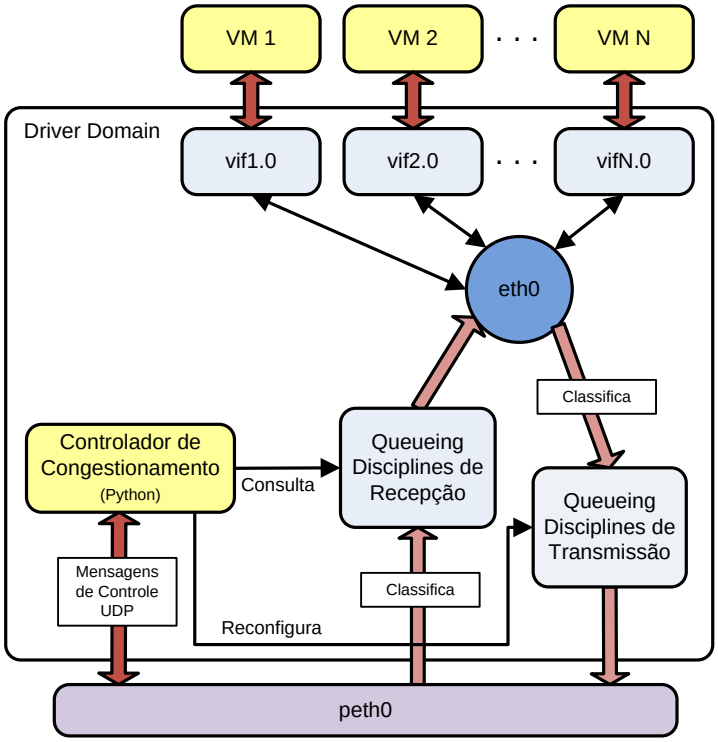


Figura 3.5. Visão geral da arquitetura do Gatekeeper em um ambiente Xen.

3.4.1 Escalonamento de transmissão

Para controlar o tráfego de transmissão das máquinas virtuais, o *Gatekeeper* usa o mecanismo de controle de tráfego fornecido pelo *kernel Linux*, discutido no Capítulo 2. O modelo de controle implementado por este mecanismo separa o tráfego que passa por uma interface de rede em uma ou mais filas, sendo cada uma controlada por um escalonador, ou política de enfileiramento, que determina a ordem na qual os dados da fila são transmitidos pela interface de rede. Dentre as diversas opções de políticas de enfileiramento oferecidas pelo *kernel Linux*, a política de enfileiramento HTB *(Hierarchical Token Bucket)* foi escolhida por suas características de organização e funcionamento, que são:

- Divisão do tráfego de rede em classes com alocações diferentes de banda passante;
- Realocação temporária de banda passante ociosa, quando uma ou mais classes de tráfego subutilizam suas alocações;
- Escalonamento de uso geral, independente das características de tráfego dos clientes.

A utilização da política HTB requer um conjunto de configurações específicas, que são aplicadas pelo agente *Gatekeeper* durante sua inicialização. Considerando que o ambiente *Xen* no qual o *Gatekeeper* é executado está configurado no modo ponte, os elementos de rede de um servidor Xen são a interface física de rede *pethX* (onde *X* depende do identificador original na interface física), a ponte em software *ethX* e as interfaces de rede virtual *vifY.Z* (onde *Y* é o identificador numérico da máquina virtual e *Z* corresponde ao número da interface de rede na máquina virtual. Por exemplo: A interface *eth2* da máquina virtual *3* é representada pela interface virtual *vif3.2*). O processo de configuração destes elementos para implementação do controle de tráfego de transmissão inclui:

- **Configurar a hierarquia de classificação na interface de rede física *pethX*:** Consiste em configurar a interface *pethX* para utilizar a política de enfileiramento HTB e criar a hierarquia de classes que divide o tráfego das máquinas virtuais.

  A construção da hierarquia de classes para o *Gatekeeper* se assemelha ao caso geral descrito no Capítulo 2: O agente cria uma classe *Raiz*, com limite próximo à capacidade do enlace, que controla a taxa global de transmissão. Em seguida, é criada uma classe para cada máquina virtual executada no servidor, com a taxa de transferência mínima sendo a taxa alocada para máquina virtual e com a taxa máxima sendo a capacidade do enlace. Desta forma, quando uma das máquinas virtuais não estiver utilizando sua alocação de rede, a banda ociosa resultante será distribuída entre as demais máquinas virtuais executadas no servidor.

  Por exemplo, a configuração de hierarquia de classes da interface de rede *peth0*, para um servidor com uma interface de rede física *peth0* e ponte *peth0*, que hospeda três máquinas virtuais, *VMA*, *VMB* e *VMC* com identificadores numéricos e alocações de banda listados pela Tabela 3.1, é mostrada pela Figura 3.6.

  Vale ressaltar que, devido à uma limitação do *Hierarchical Token Bucket*, só é possível usar cerca de 90% da capacidade total do enlace, ou 860 *Mbps*. Esta limitação, no entanto, é compensada pelos ganhos atingidos ao usar o *Gatekeeper*, como é mostrado no Capítulo 4.

- **Definir os escalonadores (políticas de enfileiramento) para cada classe de tráfego:** Após configurar as classes de tráfego das máquinas virtuais, é preciso definir quais escalonadores controlarão o envio de cada fila. Para o *Gatekeeper*, foi escolhido um escalonador *FIFO*, que escalona o envio de dados na ordem em que são enfileirados.

Tabela 3.1. Lista máquinas virtuais, identificadores e alocações de banda para o exemplo de configuração da hierarquia de classes HTB, feita pelo *Gatekeeper*

Máquina virtual	Identificador numérico	Alocação de banda
VMA	1	10%
VMB	2	30%
VMC	3	60%

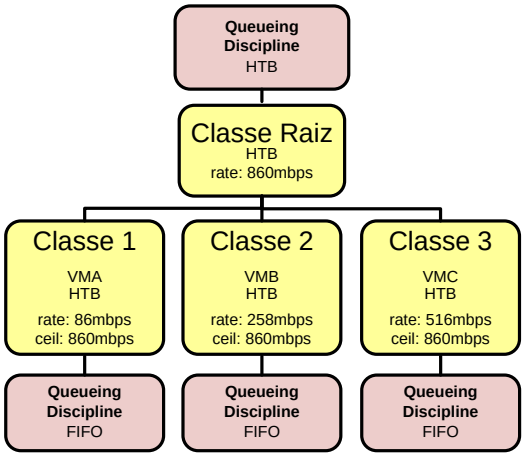


Figura 3.6. Configuração da hierarquia de classes HTB de transmissão, para a alocação da Tabela 3.1.

O escalonador FIFO utilizado pelo *Gatekeeper* foi estendido para coletar e armazenar o endereço IP de origem do último datagrama descartado por ele. Esta informação é irrelevante para o escalonamento de transmissão, mas é fundamental para a implementação do controle de congestionamento, como será elaborado na seção 3.4.3.

- **Configurar marcação de pacotes na ponte *ethX*:** Embora o mecanismo de controle de tráfego do *kernel Linux* tenha a capacidade de classificar o tráfego das máquinas virtuais observando os endereços IP de origem dos datagramas, esta solução é vulnerável a ataques nos quais um cliente malicioso pode forjar datagramas com o IP de origem modificado, para fazer suas máquinas virtuais assumirem a identidade das máquinas virtuais de outro cliente e burlarem o mecanismo de controle de tráfego.

  Para solucionar esta falha, o *Gatekeeper* filtra o tráfego de transmissão de acordo com as interfaces virtuais (*vifY.Z*) de origem, garantindo desta forma que mesmo que a máquina virtual do cliente envie datagramas forjados, eles sejam devidamente classificados e contabilizados. Esta classificação é feita pela ferramenta

iptables, que é configurada para marcar os datagramas enviados pelas máquinas virtuais à medida em que eles atravessam a ponte *ethX*.

- **Configurar os filtros de classificação:** Por último, o agente deve configurar os filtros de tráfego para classificar os datagramas que são enviados de acordo com as marcações que eles recebem ao atravessar a ponte *ethX*. Estes filtros são ligados à interface de rede *peth0*.

A Figura 3.7 mostra a configuração final do escalonamento de transmissão para o caso de exemplo. Podem ser observadas as regras de marcação na ponte *eth0* e os filtros de classificação e a hierarquia de classes na interface configurados na interface física *peth0*.

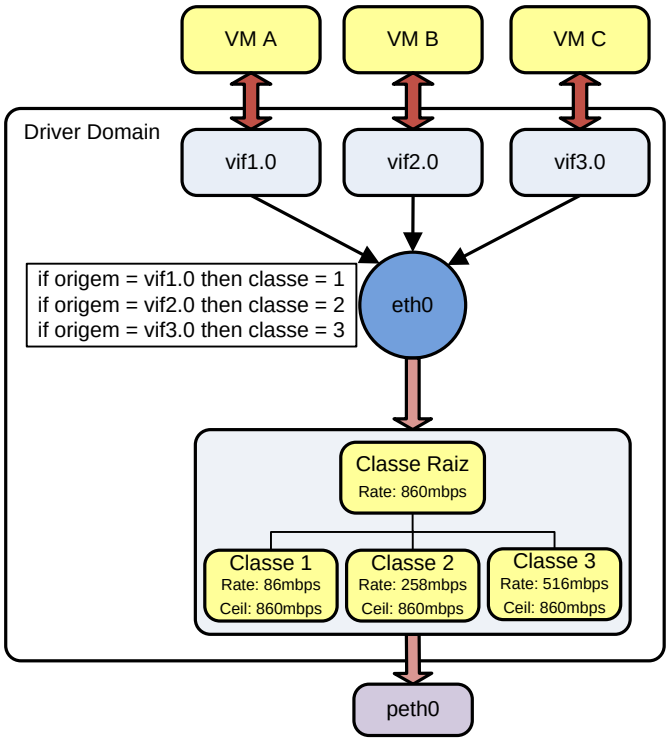


Figura 3.7. Configuração completa do escalonador de transmissão, para a alocação da Tabela 3.1.

Com isso, o procedimento *initialize_transmit_scheduler*, introduzido no Algoritmo 1, pode ser detalhado pelo Algoritmo 2. Este algoritmo é implementado por um *script* na linguagem *Bash*, invocado pelo agente *Gatekeeper*. A configuração da hierarquia de classes HTB e das políticas de enfileiramento é feita pela ferramenta TC, parte do pacote *iproute*[16] de ferramentas de configuração de rede e controle de tráfego para o *Linux*.

```
1: function initialize_transmit_schedulervms)
2: # Cria classe HTB raiz na interface peth0, com limite de envio de 860mbps
3: root_class = create_root_class(interface=peth0, limit=860mbps)
4:
5: for all virtual_machine in vms do
6:     # Cria classe de tráfego para a máquina virtual, com limite de envio igual à
       capacidade total e taxa mínima igual à sua alocação
7:     vm_class = create_leaf_class(interface=peth0, root=root_class,
       limit=virtual_machine.allocation.transmit_rate)
8:
9:     # Cria a política FIFO para a classe de tráfego da máquina virtual
10:    create_fifo_qdisc(interface=peth0, class=vm_class)
11:
12:    # Cria regra na tabela iptables da ponte eth0, que marca os pacotes das
       máquinas virtuais de acordo com a interface virtual de origem
13:    create_iptable_rule(bridge=eth0, source=virtual_machine.vif,
       assign_to_class=virtual_machine.id)
14: end for
```

Algoritmo 2: Algoritmo de inicialização do escalonador de transmissão

O escalonador de recepção será abordado na próxima seção. Sua implementação é semelhante à do escalonador de transmissão, com a adição de um mecanismo que expande as capacidades de controle de tráfego de recepção no *kernel Linux*.

3.4.2 Escalonamento de recepção

O mecanismo de controle de tráfego do *kernel Linux* também permite o controle do tráfego que chega em uma interface de rede. Contudo, ele não permite a utilização das políticas de enfileiramento de transmissão, como *HTB* e *FIFO*, disponibilizando apenas uma política, chamada *ingress*. Embora seja possível configurar filtros de classificação para esta política, seu algoritmo de escalonamento não é baseado em classes, funcionalidade necessária para a implementação do modelo de controle de tráfego do *Gatekeeper*.

Para contornar esta limitação, o *Gatekeeper* usa o módulo *IFB (Intermediary Frame Buffer)* do *kernel Linux*. Este módulo cria uma interface virtual, chamada *ifb0* por padrão. Esta interface se comporta como uma interface de rede, com a diferença de que ela repassa todos os dados que recebe para o *kernel*. Esta característica é importante para o *Gatekeeper*, porque a interface *ifb0* pode aplicar políticas de transmissão ao tráfego que repassa. Desta forma, se a interface de rede física for configurada para redirecionar seu tráfego de recepção para a interface virtual *ifb0*, é possível aplicar

políticas de transmissão a esse tráfego.

Com o uso do módulo IFB, a implementação do escalonador de recepção se assemelha à do escalonador de transmissão, com o passo adicional de configuração do módulo e do filtro de redirecionamento do tráfego que chega na interface física para a interface virtual *ifb0*. A implementação também difere na filtragem do tráfego, que é feita pelo valor do campo "destino" dos datagramas IP, ao contrário do caso de transmissão, em que a filtragem é feita pela interface virtual de origem. O algoritmo resultante é descrito pelo Algoritmo 3. A configuração gerada pelo algoritmo para as alocações listadas pela Tabela 3.1 é mostrada pela Figura 3.8.

```
1: function initialize_receive_scheduler(vms)
2: # Carrega o módulo do kernel Linux IFB, criando uma interface de rede
   intermediária ifb0
3: load_ifb_module()
4:
5: # Cria classe HTB raiz na interface peth0, com limite de envio de 860mbps
6: root_class = create_root_class(interface=ifb0, limit=860mbps)
7:
8: # Configura interface peth0 para usar a política ingress para recepção
9: set_ingress_qdisc(interface=peth0)
10:
11: for all virtual_machine in vms do
12:    # Cria classe de tráfego para a máquina virtual, com limite de recepção igual à
       capacidade total e taxa mínima igual à sua alocação
13:    vm_class = create_leaf_class(interface=ifb0, root=root_class,
       limit=virtual_machine.allocation.receive_rate)
14:
15:    # Cria a política FIFO para a classe de tráfego da máquina virtual
16:    create_fifo_qdisc(interface=ifb0, class=vm_class)
17:
18:    # Cria filtro de classificação para as máquinas virtuais, baseado em endereços
       IP
19:    create_filter(interface=ifb0, source=virtual_machine.ip,
       assign_to_class=virtual_machine.id)
20:
21:    # Cria filtro que redireciona o tráfego que chega na interface peth0 para a
       interface ifb0
22:    create_filter(interface=peth0, source=all, redirect_to=ifb0)
23: end for
```

Algoritmo 3: Algoritmo de inicialização do escalonador de recepção

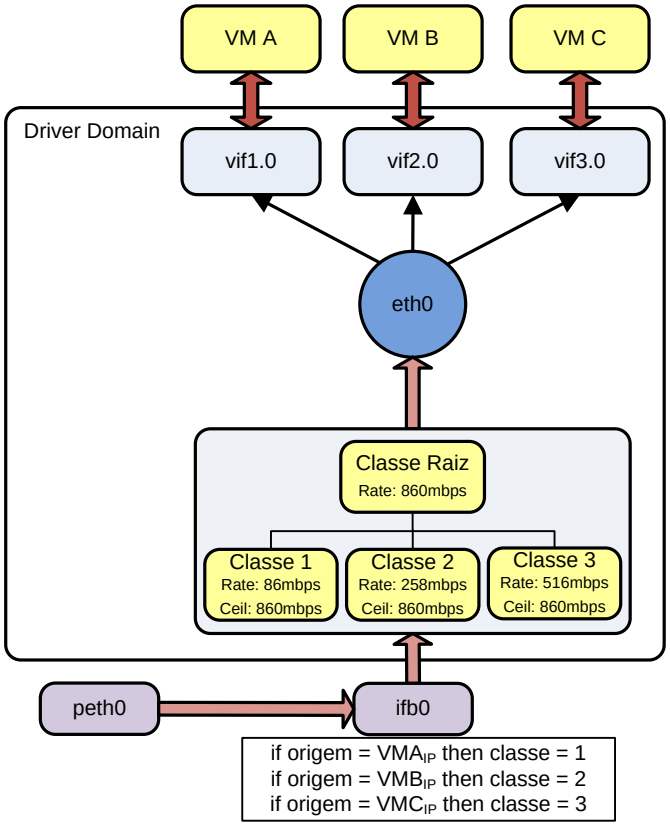


Figura 3.8. Configuração completa do escalonador de recepção, para a alocação da Tabela 3.1.

3.4.3 Controlador de congestionamento

O controlador de congestionamento tem a responsabilidade de detectar e controlar congestionamentos no tráfego de recepção das máquinas virtuais. Para isso, ele precisa coletar estatísticas das políticas de enfileiramento de recepção, para mensurar as taxas de descarte de pacotes e recepção de dados; aplicar o algoritmo de detecção de congestionamento para definir se algum transmissor deve ser controlado e atender requisições de controle que podem vir de controladores executados por outros servidores.

A coleta de estatísticas é feita através do *rtnetlink*, uma biblioteca usada para comunicação entre os subsistemas do *kernel*. Uma de suas aplicações é a modificação e consulta de parâmetros e atributos dos módulos que fazem parte do *kernel*, incluindo os módulos de escalonamento de rede, que por sua vez conhecem as políticas de enfileiramento. Para coletar as estatísticas, o controlador de congestionamento invoca um programa C, que se comunica com o módulo de escalonamento de rede através do *rtnetlink*, que retorna as estatísticas disponíveis para cada política de enfileiramento configurada.

As estatísticas utilizadas pelo algoritmo de controle de congestionamento são: as

quantidades de dados recebidos e descartados, em *bytes*, e o endereço IP de origem do último pacote descartado, que é usado para escolher a máquina virtual que deve ter sua taxa de transmissão controlada. Todavia, o módulo de escalonamento de rede armazena apenas o número de pacotes descartados, não a quantidade em *bytes* e também não guarda dados sobre o transmissor do último pacote descartado. Para disponibilizar estas informações para o controlador de congestionamento, foi criada uma versão modificada do *kernel Linux*. Essa versão, quando ocorre um descarte, atualiza uma estrutura de dados para registrar o endereço IP de origem e a quantidade dos dados, em *bytes*, descartados até então. Estes dados também são acessados através da biblioteca *rtnetlink*.

O controlador coleta as estatísticas dos escalonadores em intervalos de 10 ms, calculando a cada passo a taxa média de dados descartados durante os últimos 100 ms. Quando essa taxa ultrapassa um limiar de tolerância pré-configurado, o controlador invoca a rotina de tratamento de congestionamento. O limiar escolhido foi de 20 $Mbps$. Esta escolha significa que, em condições normais, espera-se que a taxa de dados descartados não passe de 20 $Mbps$, o que dá ao controlador de congestionamento um espaço de tempo mínimo para seguramente detectar que o tráfego não respondeu aos descartes e evita falsos-positivos. Com isso, é importante garantir que a taxa de descartes gerada por tráfego com controle de fluxo não ultrapasse este limiar. Esta garantia depende diretamente do tamanho da fila de recepção, pois como dados recebidos são descartados apenas quando a fila de recepção está cheia, o tamanho desta fila influencia diretamente as taxas de descartes observadas.

Para escolher um tamanho de fila que causasse em uma taxa de descartes apropriada para tráfego com controle de fluxo, foi conduzido um experimento em que um ou mais transmissores iniciavam múltiplas conexões TCP paralelas para um mesmo destino. O experimento foi executado para múltiplos tamanhos de fila de recepção, sendo medida a taxa máxima de descartes causada por cada combinação de número de transmissores e conexões paralelas. Os resultados para um tamanho de fila de 160 pacotes podem ser observados na Tabela 3.2 e mostram que este tamanho de fila atende a condição de gerar uma taxa de descartes de até 20 $Mbps$

Quando a taxa agregada de descartes para todas as filas de recepção ultrapassa o limiar de tolerância, o *Gatekeeper* assume que um congestionamento está ocorrendo no enlace de entrada e precisa escolher um transmissor para ser controlado. Como parâmetros para esta escolha, o *Gatekeeper* usa as taxas de recepção e de descartes das filas de cada uma das máquinas virtuais executadas pelo servidor, além das alocações de enlace de cada máquina virtual. Com estes dados, o *Gatekeeper* define o conjunto de máquinas virtuais sobrecarregadas, seguindo a Equação 3.1 e deste conjunto escolhe

Tabela 3.2. Taxas de descartes observadas pelo experimento de configuração de tamanho de fila, para uma fila de 160 pacotes.

Transmissores	Conexões TCP por transmissor	Taxa de descartes máxima
1	256	20 Mbps
4	64	16 Mbps
8	32	2.5 Mbps
16	16	5 Mbps

a fila da máquina virtual com maior diferença entre a sua taxa atual de recepção e sua alocação de enlace. Dada a fila de recepção escolhida, o transmissor selecionado para ser controlado é aquele que enviou o último pacote descartado.

Após detectar o congestionamento e definir qual transmissor que deve ser controlado, o *Gatekeeper* manda uma mensagem UDP de controle, para o agente *Gatekeeper* do servidor que executa a máquina virtual trasmissora. Os procedimentos de deteção e tratamento de congestionamentos são listados pelos Algoritmos 4 e 5.

```
1: function detect_congestion(vms, stats)
2: # O limiar de tolerância, em Mbps
3: PACKET_DROP_THRESHOLD = 20.0
4: # Variável auxiliar para armazenar a soma das taxas de descarte
5: drop_rate_sum = 0
6:
7: # Soma as taxas de descartes das filas das máquinas virtuais
8: for all virtual_machine in vms do
9:    drop_rate_sum += stats[virtual_machine].packet_drop_rate
10: end for
11:
12: # Se a taxa de descartes for maior que o limiar de tolerância, há um
    congestionamento
13: return drop_rate_sum >= PACKET_DROP_THRESHOLD
```

Algoritmo 4: Algoritmo de detecção de congestionamento

Por último, o controlador de congestionamento deve atender as requisições de agentes remotos, que podem enviar mensagens de controle para as máquinas virtuais do servidor. Quando o controlador recebe uma mensagem de controle para uma máquina virtual, ele diminui a taxa máxima de envio dessa máquina virtual pela metade. Enquanto o controlador não recebe novas mensagens de controle, ele gradualmente aumenta as taxas de transmissão das máquinas virtuais previamente controladas, até que atinjam seus valores originais. Na versão atual, a taxa de recuperação é de 1% a cada $10ms$, mas outras formas de recuperação são possíveis.

```
1: function handle_congestion(vms, stats)
2: # Variável que guarda a fila sobrecarregada
3: overloaded_queue = None
4: max_phi = 0
5:
6: for all virtual_machine in vms do
7:    # Calcula a diferença entre a taxa de recepção e a alocação da máquina virtual
8:    phi = stats[virtual_machine].receive_rate - virtual_machine.allocation
9:
10:   # A fila da máquina virtual com maior diferença é dada como sobrecarregada
11:   if phi > max_phi then
12:      max_phi = phi
13:      overloaded_queue = virtual_machine.queue
14:   end if
15: end for
16:
17: if overloaded_queue is not None then
18:   # Transmissor escolhido é origem do último pacote descartado
19:   source = overloaded_queue.last_drop_source
20:
21:   # Se o transmissor é uma máquina virtual, enviar mensagem de controle para
      agente remoto
22:   if is_virtual_machine(source) then
23:      send_control_message(source)
24:   end if
25: end if
```

Algoritmo 5: Algoritmo de tratamento de congestionamento

Concluída a descrição da implementação do *Gatekeeper*, é necessário mostrar que o sistema divide corretamente o enlace entre as máquinas virtuais e que detecta e controla congestionamentos no enlace de entrada. O próximo capítulo descreve os experimentos construídos para avaliar o funcionamento do *Gatekeeper*, discutindo seus resultados.

Capítulo 4

Avaliação

No capítulo anterior, foi descrita a implementação do protótipo do *Gatekeeper*, que é executado pelo *Driver Domain* como um processo em nível de usuário. O agente configura os mecanismos de controle de tráfego do *kernel Linux*, coletando suas estatísticas para detectar e mitigar congestionamentos no enlace de entrada. O foco deste capítulo é a avaliação deste agente, na qual foram medidas a exatidão do seu mecanismo de divisão de tráfego e seu impacto no desempenho de uma aplicação *Hadoop*, na presença de tráfego TCP e UDP concorrente.

As próximas seções detalharão os experimentos de avaliação e apresentarão seus resultados, mostrando que, além de corretamente aplicar as garantias de taxas de transmissão e recepção de dados, o *Gatekeeper* aprimora o desempenho de aplicações que concorrem com tráfego sem controle de fluxo.

4.1 Avaliação de exatidão

Nesta avaliação, o desempenho de divisão de tráfego do *Gatekeeper* foi comparado com o mecanismo de controle de tráfego padrão do *Linux*. Este mecanismo foi testado em duas configurações: a primeira, é permite a divisão da capacidade de enlace ociosa entre os clientes, enquanto a segunda impõe limites fixos de transmissão e recepção e não permite a reutilização de recursos ociosos. Estas duas configurações serão identificadas como CTPR[1] e CTPLF[2].

A avaliação usou uma configuração na qual três servidores físicos, $H1$, $H2$ e $H3$, executam máquinas virtuais de dois clientes diferentes, A e B. O servidor $H1$ executa uma máquina virtual por cliente: VMA_1 para o cliente A e VMB_1 para o cliente

[1]Controle de Tráfego Padrão com Reuso
[2]Controle de Tráfego Padrão com Limites Fixos

B, enquanto o servidor $H2$ executa uma máquina virtual VMA_2 para o cliente A e o servidor $H3$ uma máquina virtual VMB_2 pertencente ao cliente B. Nesta configuração, cada cliente executa um *microbenchmark netperf* entre suas duas máquinas virtuais por 10 segundos, sendo examinados dois cenários:

- **Cenário 1:** As máquinas virtuais do servidor $H1$ iniciam transmissões para as máquinas virtuais de seus respectivos clientes nos servidores $H2$ e $H3$. Neste caso, o gargalo de rede está no enlace de transmissão em $H1$;
- **Cenário 2:** As máquinas virtuais dos servidores $H2$ e $H3$ iniciam transmissões para as máquinas virtuais de seus respectivos clientes no servidor $H1$. Neste caso, o gargalo de rede está no enlace de recepção em $H1$.

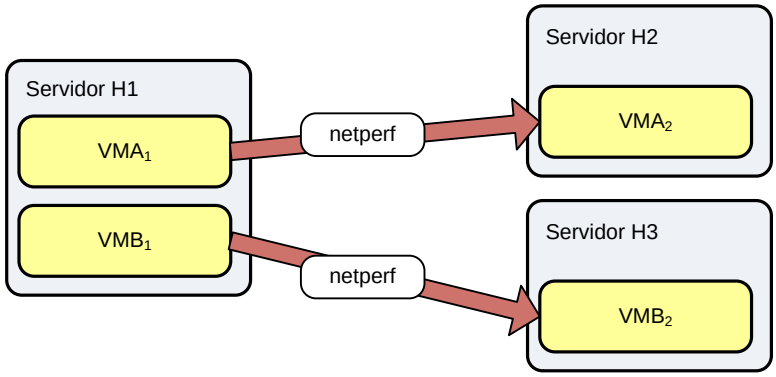
O objetivo destes dois cenários é avaliar a corretude do algoritmo para uma configuração mínima, avaliando casos nos quais tráfego com controle de fluxo concorre pelo enlace con tráfego com e sem controle de fluxo. que garante que o o tráfego concorrente não sofrerá interferência. Sem interferência, os efeitos da falta de controle de fluxo serão potencializados, evidenciando a necessidade de um método de controle de fluxo independente de tipo de tráfego.

A Figura 4.1 ilustra os dois cenários examinados. Foram executados seis testes em cada cenário, nos quais são iniciadas diferentes tipos de transmissões, com e sem tráfego de fluxo, entre as máquinas virtuais dos clientes. As combinações de fluxos das transmissões, para cada teste, são listadas pela Tabela 4.1.

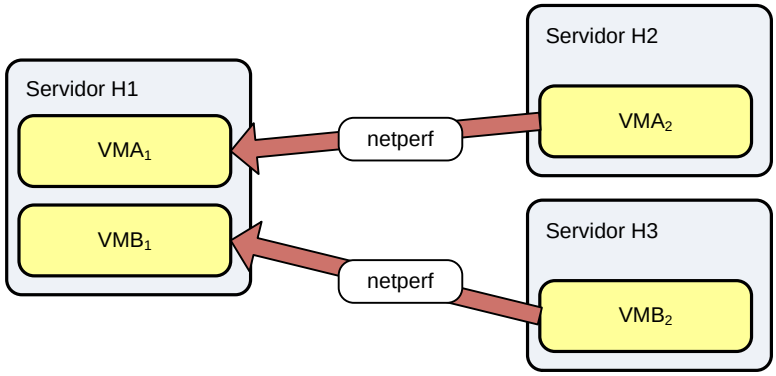
Tabela 4.1. Combinações de fluxos usadas nos testes, onde VMA_t e VMB_t são as máquinas virtuais transmissoras nos dois cenários.

Teste	Fluxo gerado por VMA_t	Fluxo gerado por VMB_t
1	1 conexão TCP	Nenhum fluxo
2	1 conexão TCP	1 conexão TCP
3	1 conexão TCP	10 conexões TCP
4	1 conexão TCP	1 fluxo UDP
5	1 fluxo UDP	Nenhum fluxo
6	1 fluxo UDP	1 fluxo UDP

As alocações de enlace para os clientes foram definidas como 75% para o cliente A e 25% para o cliente B. Devido a uma limitação no mecanismo de controle de tráfego do *kernel Linux*, discutida no Capítulo 2, o limite total para as taxas de envio e recepção usado foi de 860 $Mbps$, resultando em alocações de 645 $Mbps$ para o cliente A e 215 $Mbps$ para o cliente B.



(a) Cenário 1

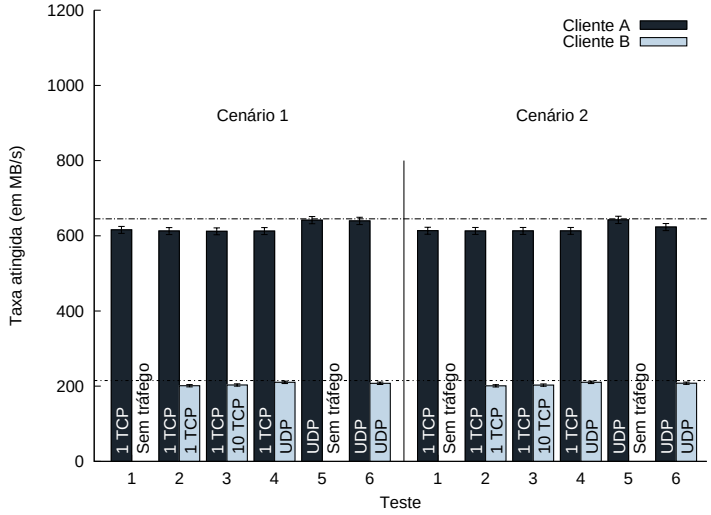


(b) Cenário 2

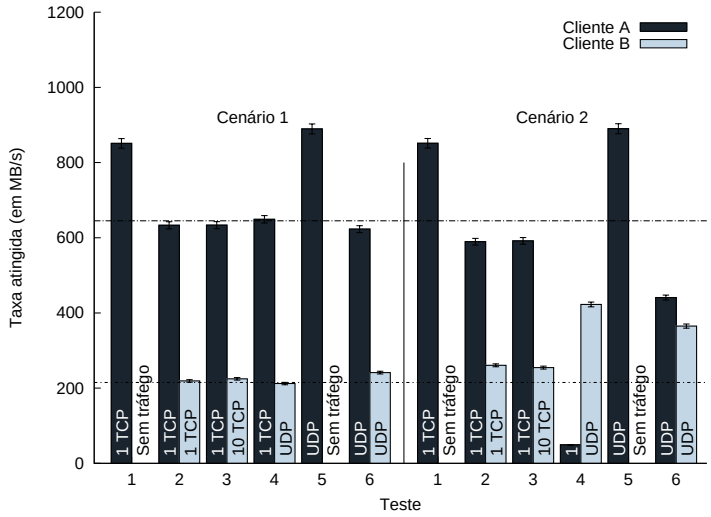
Figura 4.1. Cenários de transmissão e recepção da avaliação de exatidão.

Cada cenário foi executado cinco vezes. Os resultados da avaliação, com barras de erro para um intervalo de confiança de 95%, podem ser observados na Figura 4.1. Os resultados para os dois clientes estão agrupados por testes e por cenários, com as barras escuras reportando as taxas de transmissão alcançadas pelas máquinas virtuais do cliente A e as claras, as taxas atingidas pelas máquinas do cliente B. As linhas pontilhadas referenciam as alocações dos clientes, sendo a linha superior a alocação do cliente A (645 $Mbps$) e a inferior a alocação do cliente B (215 $Mbps$).

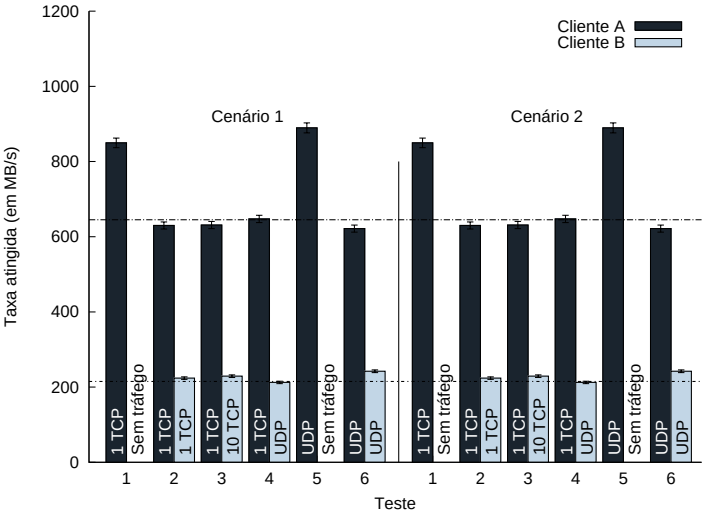
A Figura 4.2(a) mostra os resultados da avaliação quando utiliza-se o mecanismo de controle de tráfego padrão, sem realocação de banda. Os resultados mostram que o mecanismo divide corretamente os recursos entre os clientes, embora exista uma perceptível degradação de desempenho. Além desta degradação, nota-se que, para os testes 1 e 5 de ambos cenários, mesmo quando os recursos do cliente B estão ociosos o cliente A atinge uma banda próxima a sua alocação. Assim, embora tal configuração aplique corretamente as alocações configuradas para ambos cenários, ela desperdiça a capacidade do enlace quando apenas um fluxo é presente, além de impor uma ligeira degradação de desempenho.



(a) Controle de tráfego com limites fixo (CTPLF)



(b) Controle de tráfego com reuso de banda (CTPR)



(c) Gatekeeper

Figura 4.2. Resultados para o teste de exatidão

Dados os resultados para o mecanismo de controle de tráfego padrão sem realocação, um raciocínio inicial seria de que o mecanismo com realocação de banda resolveria o problema de desperdício de recursos, realocando os recursos do cliente B para o cliente A. Contudo, os resultados mostrados na Figura 4.2(b) mostram que, embora habilitar a realocação de banda produza resultados melhores para os testes 1 e 5, quando apenas o cliente A gera tráfego, o mesmo não vale para os outros testes no Cenário 2. Neste cenário, o mecanismo de controle de tráfego padrão do *kernel Linux* impõe uma severa degradação de desempenho para o cliente A no Teste 4, no qual um fluxo TCP, iniciado pela máquina virtual VMA_2 compete com um fluxo UDP iniciado pela máquina virtual VMB_2. Além disso, quando dois fluxos UDP competem entre si pelo enlace de recepção (Teste 6) no segundo cenário, o mecanismo de controle de tráfego falha ao aplicar as alocações de banda, pois ambos não respondem às perdas de pacotes que causam e não ajustam suas taxas de transmissão.

Quando utiliza-se o *Gatekeeper* (Figura 4.2(c)), não apenas o enlaces de transmissão e recepção são corretamente alocados, como também é permitida a realocação da capacidade ociosa do enlace, o que mostra que o *Gatekeeper* consegue unir as vantagens de ambas configurações do mecanismo padrão. Os resultados, no entanto, mostram uma ligeira degradação de desempenho de recepção em relação aos resultados para o mecanismo padrão sem realocação de capacidade. Tal degradação é uma contrapartida aceitável quando contraposta com os ganhos obtidos ao evitar o desperdício da capacidade ociosa do enlace.

Afirmada a eficiência do *Gatekeeper* em dividir o enlace de transmissão e recepção, realocando a capacidade ociosa entre os clientes, se faz necessário avaliar o seu impacto na execução de uma aplicação real. A próxima seção apresenta a avaliação do efeito do *Gatekeeper* na execução de uma aplicação *Hadoop*, executado por um cliente, quando esta compete com tráfego TCP ou UDP iniciado por outros clientes.

4.2 Avaliação de impacto no desempenho de uma aplicação *Hadoop*

Os resultados anteriores mostram que o *Gatekeeper* aplica as alocações do enlace corretamente, para cenário estático envolvendo um número pequeno de máquinas. O passo seguinte foi avaliá-lo em um ambiente mais realista, usando um maior número de máquinas e uma carga que apresentasse demanda de rede que variasse com o tempo. O objetivo deste experimento foi avaliar o comportamento do *Gatekeeper* ao controlar aplicações com demanda dinâmica de rede, que concorrem com tráfego com e sem con-

trole de fluxo. Para implementar tal aplicação com demanda dinâmica, foi escolhido o ambiente *Hadoop* de programação distribuída.

O *Hadoop* [7] é uma implementação de código aberto do paradigma de programação *MapReduce* [10], que simplifica e automatiza a execução de aplicações paralelas em ambientes de *clusters*. No paradigma *MapReduce*, a execução de uma aplicação é dividida em duas fases: *Map*, em que múltiplos conjuntos de entrada são processados independentemente por máquinas separadas e *Reduce*, na qual as saídas resultantes das execuções da fase *Map* são agregadas por uma máquina do *cluster*. Como, durante a execução de uma aplicação *Hadoop*, as máquinas do *cluster* precisam se comunicar através da rede para transferir dados ou coordenar a execução da aplicação, este sistema foi escolhido como um exemplo para a avaliação do impacto do *Gatekeeper* no desempenho de aplicações de rede. Assim, espera-se que se o mecanismo de controle de tráfego não aplicar corretamente as alocações de enlace, o desempenho da aplicação Hadoop sofrerá degradação.

No cenário avaliado, uma aplicação *Hadoop* é executada em um *cluster* de 26 máquinas virtuais, distribuídas em 26 servidores. A aplicação *Hadoop* executa uma carga personalizada, chamada "Random", cuja fase *Map* consiste em gerar dados aleatórios a partir de um conjunto de arquivos de entrada, e a fase *Reduce* em contar o número de linhas geradas. Esta carga, embora simples, gera uma demanda de rede dinâmica significativa, causada pelas transferências de dados entre as fases *Map* e *Reduce*, que se estende por um período de tempo longo o suficiente para garantir que o desempenho da aplicação é sensível à disponibilidade dos recursos de rede. Em paralelo, são executadas outras 26 máquinas virtuais, colocadas com as máquinas do *cluster Hadoop*, que executam *microbenchmarks netperf* entre si, formando 13 pares de transmissores e receptores, com cargas formada por um fluxo UDP ou 10 conexões TCP paralelas por par. O ambiente dos experimentos é ilustrado pela Figura 4.3.

Foram avaliados o desempenho da aplicação *Hadoop* e dos *microbenchmarks* concorrentes para ambientes com as mesmas configurações da análise de exatidão: com o mecanismo de controle de tráfego padrão do *Linux* com realocação de enlace ocioso, com limites fixos e com o *Gatekeeper*. Adicionalmente, foi avaliado o caso em que não é utilizada qualquer forma de controle de tráfego. Em termos de alocação de enlace, foram aplicadas três configurações, variando a alocação da aplicação *Hadoop* entre 25%, 50% e 75%, reservando o complemento para o tráfego concorrente. O resultado esperado é que, quanto maior a alocação para a aplicação *Hadoop*, menor será seu tempo de execução e menor será a banda de transmissão atingida pelo tráfego concorrente. Neste contexto, o objetivo do *Gatekeeper* é minimizar o tempo de execução da aplicação *Hadoop*, enquanto maximiza a taxa de transferência do tráfego concorrente.

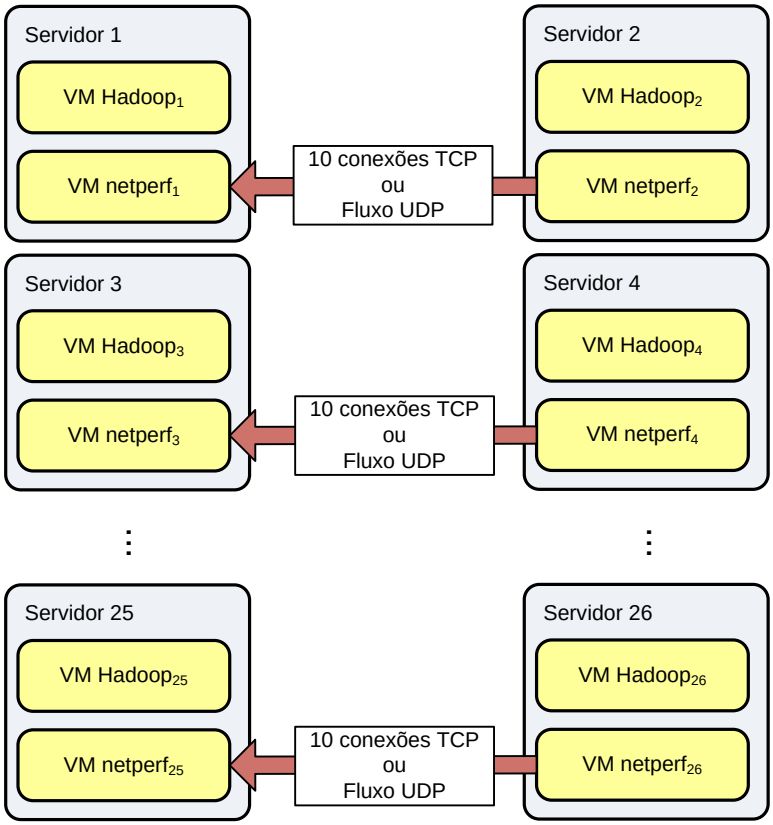


Figura 4.3. Cenário da avaliação do impacto de desempenho do *Gatekeeper*. Máquinas virtuais *Hadoop* compartilham recursos com máquinas virtuais que executam *microbenchmarks* TCP e UDP.

Os experimentos foram executados cinco vezes, com resultados para um intervalo de confiança de 95% mostrados pelas Figuras 4.4 e 4.5. As Figuras 4.4(a) e 4.5(a) apresentam os tempos de execução da aplicação *Hadoop* para as diferentes configurações de controle de tráfego e tipos de fluxo, enquanto as Figuras 4.4(b) e 4.5(b) apresentam as taxas de transmissão médias alcançadas pelos *microbenchmarks* concorrentes. Nos gráficos de tempo de execução, as linhas pontilhadas se referem ao tempo de execução obtido quando a aplicação *Hadoop* é executada sem tráfego concorrente, ou seja, o melhor resultado possível.

4.2.1 Aplicação *Hadoop* x Tráfego TCP

Os resultados para os testes com tráfego concorrente, formado por 10 conexões TCP por par de transmissor/receptor concorrente, apontam que este causa uma perda significativa de desempenho quando nenhuma forma de controle de tráfego é utilizada, causando um aumento no tempo de execução de 41 s para 75 s, um aumento de quase 83%. Isso se dá porque, como não há qualquer forma de controle de tráfego, o en-

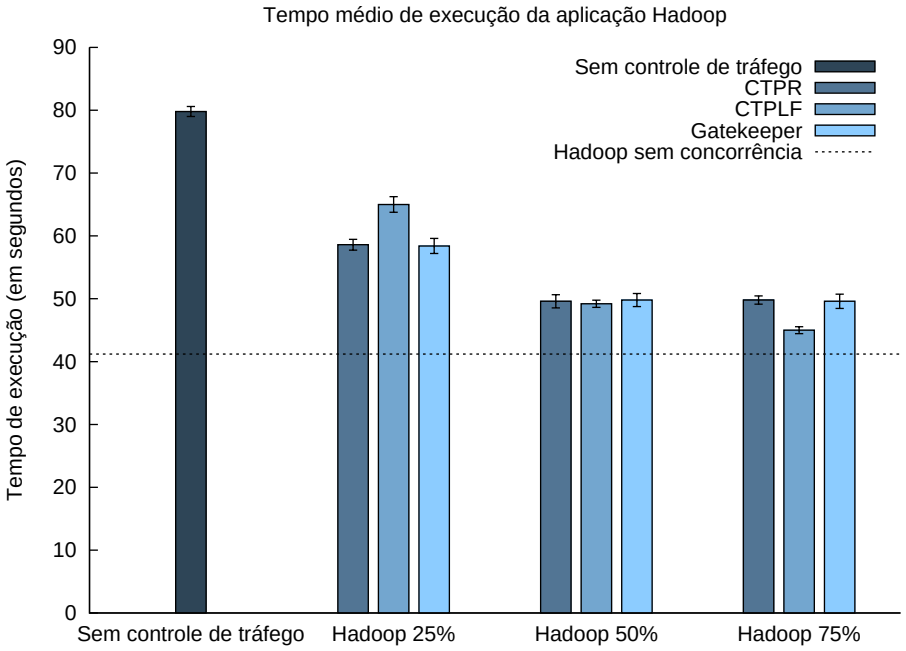
lace tende a ser dividido igualmente entre cada conexão TCP. Desta forma, o fluxo concorrente utiliza uma fatia maior do enlace por iniciar mais conexões paralelas.

Quando habilita-se um mecanismo de controle de tráfego o resultado é, em linhas gerais para todas as alternativas, um decréscimo no tempo de execução da aplicação *Hadoop* e também na taxa de transferência atingida pelo tráfego TCP, que varia de acordo com a alocação de enlace para a aplicação (quanto maior a alocação para aplicação, menor seu tempo de execução e menor a alocação para o tráfego concorrente). Isto acontece porque, como o tráfego TCP se adapta aos limites impostos pelos mecanismos, a aplicação *Hadoop* consegue utilizar uma porção maior do enlace, em detrimento do fluxo TCP concorrente. Comparando os mecanismos entre si, nota-se que o desempenho do *Gatekeeper* combina os melhores aspectos dos mecanismos de controle padrão com e sem realocação de enlace ocioso, como será discutido a seguir.

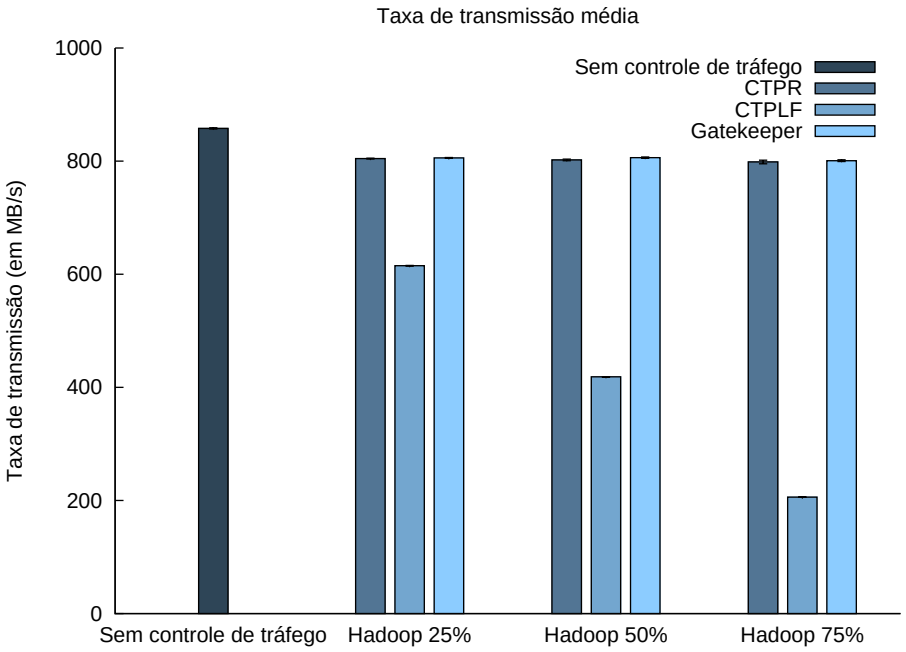
A comparação entre os mecanismos envolve duas métricas básicas: tempo de execução da aplicação *Hadoop* e taxa de transferência alcançada pelo tráfego concorrente. Em termos de tempo de execução e taxa de transferencia, o desempenho do *Gatekeeper* é similar ao do mecanismo padrão com realocação de enlace, como visto na Figura 4.4(a), pois ambos dividem corretamente o tráfego com controle de fluxo e, quando a aplicação *Hadoop* não usa toda sua porção de enlace (antes e depois da transição entre as fases *Map* e *Reduce*), o realocam para o tráfego concorrente.

No entanto, tanto o *Gatekeeper* quanto o mecanismo padrão com realocação de enlace mostram um degradação perceptível no desempenho da aplicação *Hadoop*, evidenciado por um acréscimo de cerca de 8% no seu tempo de execução quando a alocação de enlace para a aplicação é de 75%, em comparação com o mecanismo padrão com limites fixos. Neste caso, as políticas de descartes mais agressivas do *Gatekeeper* e do mecanismo padrão com realocação causam um número maior de descartes nos enlaces de entrada, devido às taxas de envio mais altas do tráfego concorrente. Consideramos que esta degradação, por outro lado, é aceitável dados os ganhos na taxa de transferência do tráfego concorrente, observados na Figura 4.4(b), que vão desde cerca de 30%, quando a alocação do enlace é de 25% para a aplicação *Hadoop*, até por volta de 288%, quando a alocação é de 75%.

O resultado desta fase da avaliação mostrou que o *Gatekeeper*, quando controla apenas tráfego com controle de fluxo, se comporta de forma semelhante à do mecanismo padrão com realocação de enlace ocioso, que mantém o desempenho da aplicação enquanto maximiza o uso do enlace. Na próxima seção, serão apresentados os resultados da avaliação quando o tráfego concorrente é governado pelo protocolo UDP, que não possui controle de fluxo.



(a) Tempo de execução da aplicação *Hadoop*



(b) Taxas de transmissão dos fluxos concorrentes

Figura 4.4. Resultados para o teste com uma aplicação *Hadoop* competindo com um fluxo formado por 10 conexões TCP.

4.2.2 Aplicação *Hadoop* x Tráfego UDP

Quando o tráfego concorrente não possui uma forma de controle de fluxo, mecanismos baseados em reação a descartes, como o mecanismo de controle padrão do *kernel Linux*, são ineficazes em controlar as taxas de recepção. A Figura 4.5(b) mostra que, ao usar o mecanismo padrão, as taxas de recepção seguem as alocações apenas quando as taxas de transmissão possuem limites fixos, sem realocação de enlace. O objetivo desta segunda parte da avaliação é mostrar que o mecanismo de controle de congestionamento do *Gatekeeper* supera esta limitação do mecanismo de controle padrão com realocação, permitindo o controle de fluxos não responsivos enquanto mantém a eficiência proporcionada pela realocação de enlace.

Assim como na avaliação com tráfego TCP, há duas métricas principais: o tempo de execução da aplicação *Hadoop* e a taxa de transferência atingida pelo tráfego concorrente que, neste caso, é formado por um fluxo UDP. Observando os tempos de execução (Figura 4.5(a)), quando a aplicação *Hadoop* compete com tráfego formado por um fluxo UDP entre as máquinas virtuais concorrentes, observa-se na uma perda acentuada de desempenho da aplicação quando utiliza-se o mecanismo de controle de tráfego padrão com realocação de enlace. Isto acontece porque quando o mecanismo tenta controlar as taxas de recepção descartando pacotes, o fluxo UDP não reage, enquanto que as conexões TCP usadas pela aplicação reagem. A consequência disto é um fenômeno em que as conexões TCP diminuem suas taxas para liberar o enlace de recepção, criando um espaço que é rapidamente utilizado pelo fluxo UDP, o que gera mais descartes, criando uma reação em cadeia. O dados da avaliação mostram que a degradação de desempenho da aplicação chega a 57%, quando a alocação de enlace para a aplicação é de 75%.

A introdução do mecanismo de controle de congestionamento do *Gatekeeper*, por sua vez, produz resultados próximos ou melhores do que os alcançados com o mecanismo de controle padrão com limites fixos, o que reforça a premissa de que o *Gatekeeper* divide corretamente os recursos do enlace. A Figura 4.5(b) aponta que a introdução do mecanismo de controle de congestionamento do *Gatekeeper* não afeta negativamente o desempenho do tráfego concorrente, produzindo resultados melhores do que os do mecanismo padrão com realocação. Isto acontece porque o mecanismo de controle de congestionamento do *Gatekeeper* diminui a ocorrência de descartes de dados no enlace de entrada, promovendo menor desperdício do enlace de entrada.

Estes resultados comprovam que os mecanismos atuais de controle de tráfego de rede ora desperdiçam recursos ao impor limites fixos de transmissão e recepção, ora falham em alocar corretamente os recursos entre os diferentes clientes, quando parte

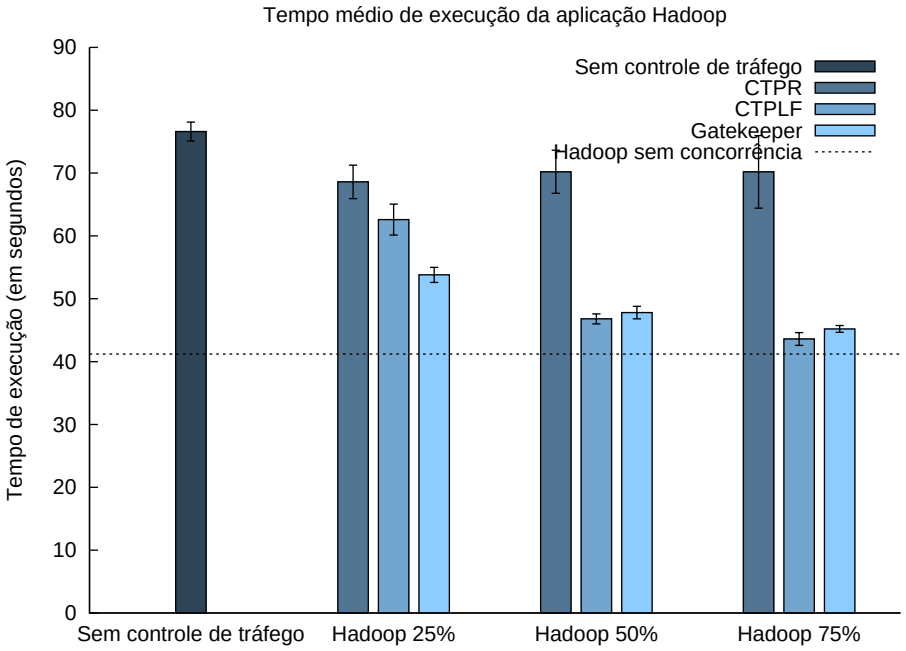
do tráfego de rede é não implementa alguma forma de controle de fluxo. Esta falha é inerente à natureza passiva dos mecanismos de controle de recepção, que se limitam a descartar parte dos dados que recebem, esperando que os transmissores reajam a esses descartes e limitem suas taxas de transmissão.

Embora tal comportamento seja suficiente quando o tráfego dos transmissores possui controle de fluxo, como o de conexões TCP, o mesmo não pode ser aplicado quando os transmissores geram tráfego sem esse controle, como fluxos UDP, que ignoram os descartes feitos pelo receptor. O *Gatekeeper*, ao introduzir um mecanismo de controle de congestionamento na camada de virtualização, elimina esta desvantagem, permitindo a realocação de enlace ocioso entre as máquinas virtuais do servidor, sem comprometer a divisão de recursos configurada para os clientes.

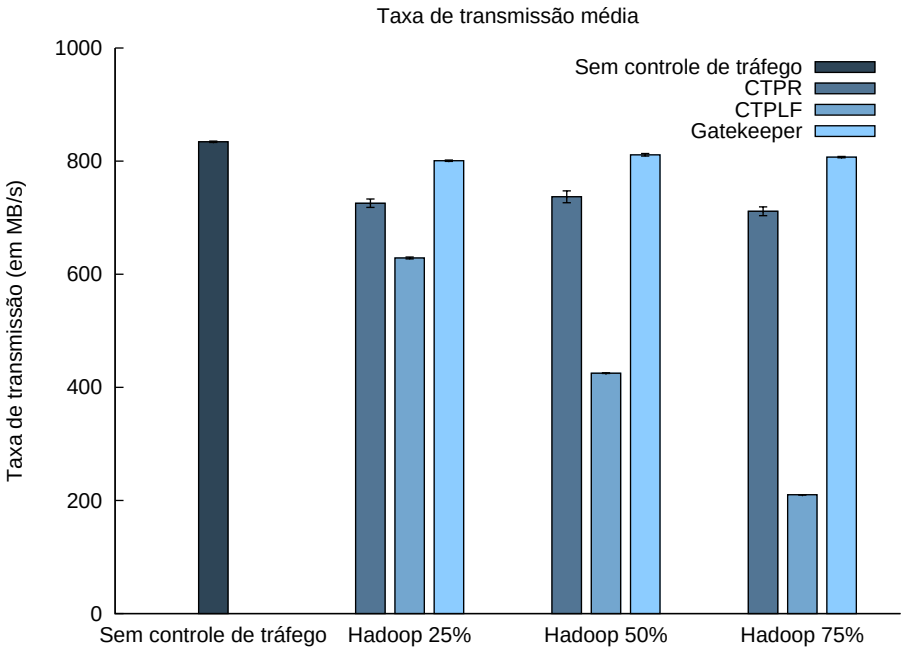
4.3 Considerações finais

Este capítulo mostrou os experimentos conduzidos para avaliar a exatidão do mecanismo de alocação de recursos do *Gatekeeper* e o seu impacto na execução de aplicações de rede, quando estas competem com tráfego UDP, que não possui controle de fluxo. Os resultados apontaram que o *Gatekeeper* dividiu os recursos de rede corretamente entre os clientes, mesmo em situações em que tráfego TCP competia com tráfego UDP e que, em um experimento em que uma aplicação *Hadoop* competia com tráfego UDP, o *Gatekeeper* não apenas diminui significativamente o tempo de execução da aplicação, como também aumentou sensivelmente a taxa de transmissão média alcançada pelo tráfego concorrente.

Estes resultados mostram que o mecanismo proposto cumpre suas funções de forma satisfatória, provendo um forte argumento para o uso da camada de virtualização para aprimorar os mecanismos atuais de alocação de recursos de rede.



(a) Tempo de execução da aplicação *Hadoop*



(b) Taxas de transmissão dos fluxos concorrentes

Figura 4.5. Resultados para o teste com uma aplicação *Hadoop* competindo com um fluxo UDP.

Capítulo 5

Conclusão e trabalhos futuros

A massificação do paradigma de "computação-como-serviço", realizado na forma do modelo de *Cloud Computing*, aumentou de forma significativa a demanda de recursos nos *datacenters* em que tais serviços são implementados. Este aumento de demanda sublinha a necessidade maximizar a eficiência da utilização dos recursos do *datacenter*, que fazem uso de ambientes de virtualização para permitir que múltiplas maquinas virtuais, potencialmente de clientes diferentes, sejam executadas por um mesmo servidor. Ao dividir os recursos de uma mesma máquina física entre máquinas virtuais distintas, é essencial garantir o isolamento de recursos físicos entre elas, seja para evitar falhas de segurança ou para atender requisitos contratuais de desempenho.

Embora as tecnologias de divisão de recursos de processamento e memória estejam consolidadas, o mesmo não pode ser dito sobre a divisão de recursos de rede. Os mecanismos de controle de tráfego atuais ora dependem de soluções específicas de *hardware*, ora foram projetadas para o controlar tráfego gerado por protocolos com controle de fluxo, como TCP, falhando em controlar protocolos que não possuem tal controle, como UDP. Isto impossibilita a aplicação de políticas mais abrangentes de alocação de recursos, pois as soluções atuais ora dependem de *hardware* específico ora tratam apenas de tráfego com controle de fluxo.

Com o objetivo de oferecer controle de tráfego de rede confiável e eficiente no ambiente de *datacenters* virtualizados, foi desenvolvido o *Gatekeeper*, um mecanismo distribuído de controle de tráfego de rede para ambientes virtualizados. A proposta do *Gatekeeper* é utilizar o nível de abstração adicional criado pela camada de virtualização, para implementar um mecanismo de detecção e controle de congestionamentos no enlace de entrada. Este mecanismo, independente de protocolo, observa as taxas de descartes no enlace de entrada do servidor, notificando os servidores transmissores quando esta taxa de descartes ultrapassa um limiar de tolerância pré-configurado.

Quando um servidor recebe uma notificação de congestionamento, ele diminui a taxa de transmissão da máquina virtual identificada como causadora do congestionamento, gradualmente recuperando sua taxa de transmissão enquanto não receber notificações adicionais.

A avaliação da exatidão de divisão de recursos de rede mostrou que o *Gatekeeper*, além de alocar corretamente os enlaces de rede de transmissão e recepção entre as máquinas virtuais de um servidor, utiliza o enlace de forma eficiente, realocando recursos de enlace ociosos entre os clientes com demanda. Também foi avaliado o impacto do *Gatekeeper* na execução de uma aplicação distribuída Hadoop, executada em um *cluster* de máquinas virtuais que dividia recursos com um conjunto de máquinas virtuais colocadas, que geravam tráfego TCP ou UDP durante a execução da aplicação. Os resultados desta segunda avaliação confirmaram que, na ausência do mecanismo de controle de congestionamento do *Gatekeeper*, o desempenho da aplicação é perceptivelmente afetado pelo tráfego concorrente. Com o uso do *Gatekeeper*, além de ser observada a melhoria do desempenho da aplicação, também foi registrado o aumento das taxas de transmissão alcançadas pelo tráfego concorrente, o que reafirma que o *Gatekeeper* mantém ou melhora o desempenho das aplicações ao mesmo tempo que maximiza o uso do enlace de rede.

Mesmo com estes resultados positivos, ainda há oportunidades de aprimorar o mecanismo e expandir seu escopo. Os trabalhos futuros sugeridos são a implementação de um algoritmo de prevenção de congestionamento, que analise as taxas de recepção das máquinas virtuais dos servidores e tente controlar as taxas de envio dos transmissores antes que eles causem congestionamentos e uma modelagem mais abrangente do mecanismo de recuperação de taxas de transmissão, que neste trabalho usa um conjunto de parâmetros estáticos, para permitir uma reconfiguração dinâmica que aprimore a eficiência do mecanismo.

Referências Bibliográficas

[1] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. In *Proceedings of the ACM 2008 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)*, pages 63–74, Seattle, WA, Agosto de 2008.

[2] W. Almesberger and E. ICA. Linux Network Trac Control: Implementation Overview. *White Paper*, Abril de 1999.

[3] W. Almesberger, E. Ica, J. Salim, A. Kuznetsov, and I. Moscow. Differentiated services on Linux. In *In Proceedings of GlobeCom*, 1999.

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

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

[6] J. Bennett and H. Zhang. WF^ 2Q: Worst-case fair weighted fair queueing. In *IEEE INFOCOM*, volume 96, pages 120–128. INSTITUTE OF ELECTRICAL ENGINEERS INC (IEEE), 1996.

[7] A. Bialecki, M. Cafarella, D. Cutting, and O. O'Malley. Hadoop: a framework for running applications on large clusters built of commodity hardware. 2005.

[8] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker. Ethane: taking control of the enterprise. In *Proceedings of the ACM 2007 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)*, pages 1–12, Kyoto, Japan, Agosto de 2007.

[9] B. Clark, T. Deshane, E. Dow, S. Evanchik, M. Finlayson, J. Herne, and J. N. Matthews. Xen and the art of repeated research. In *Proceedings of the Usenix annual technical conference, Freenix track*, pages 135–144, 2004.

[10] J. Dean and S. Ghemawat. Map Reduce: Simplified data processing on large clusters. *Communications of the ACM-Association for Computing Machinery-CACM*, 51(1):107–114, 2008.

[11] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queueing algorithm. In *Symposium proceedings on Communications architectures & protocols*, pages 1–12. ACM, 1989.

[12] B. Des Ligneris. Virtualization of Linux based computers: the Linux-VServer project. In *High Performance Computing Systems and Applications, 2005. HPCS 2005. 19th International Symposium on*, pages 340–346, 2005.

[13] M. Devera. Hierarchical token bucket theory, Fevereiro de 2010. `http://luxik.cdi.cz/~devik/qos/htb/manual/theory.htm`.

[14] N. Duffield, P. Goyal, A. Greenberg, P. Mishra, K. Ramakrishnan, and J. van der Merive. A flexible model for resource management in virtual private networks. In *Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communication*, pages 95–108. ACM New York, NY, USA, 1999.

[15] EC2: Amazon Elastic Compute Cloud, Janeiro de 2010. `http://aws.amazon.com/ec2/`.

[16] T. L. Foundation. iproute2: The Linux Foundation, Fevereiro de 2010. `http://www.linuxfoundation.org/collaborate/workgroups/networking/iproute2`.

[17] S. Golestani and M. Bellcore. A self-clocked fair queueing scheme for broadband applications. In *INFOCOM'94. Networking for Global Communications., 13th Proceedings IEEE*, pages 636–646, 1994.

[18] Google App Engine, Janeiro de 2010. `http://code.google.com/appengine/`.

[19] A. Greenberg, J. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. Maltz, P. Patel, and S. Sengupta. VL2: A scalable and flexible data center network. *ACM SIGCOMM Computer Communication Review*, 39(4):51–62, 2009.

[20] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. VL2: A scalable and flexible data center network. In *Proceedings of the ACM SIGCOMM 2009 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, pages 51–62, Barcelona, Spain, Agosto de 2009.

[21] A. Greenberg, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. 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)*, pages 57–62, Seattle, WA, 2008.

[22] L. Grinzo. Getting Virtual with VMware 2.0. *Linux Magazine*, June 2000.

[23] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker. Nox: towards an operating system for networks. *Computer Communication Review*, 38(3):105–110, 2008.

[24] C. Guo, G. Lu, D. Li, H. Wu, X. Zhang, C. T. Yunfeng Shi, Y. Zhang, and S. Lu. BCube: A high performance, server-centric network architecture for modular data centers. In *Proceedings of the ACM SIGCOMM 2009 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, pages 63–74, Barcelona, Spain, Agosto de 2009.

[25] C. Guo, H. Wu, K. Tan, L. Shi, Y. Zhang, and S. Lu. DCell: A scalable and fault-tolerant network structure for datacenters. In *Proceedings of the ACM SIGCOMM 2008 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, pages 75–86, Seattle, WA, Agosto de 2008.

[26] B. Hubert, T. Graf, G. Maxwell, R. van Mook, M. van Oosterhout, and P. Schroeder. Linux advanced routing and traffic control. In *Ottawa Linux Symposium*, pages 213–222, 2003.

[27] IFB: The Linux Foundation, Fevereiro de 2010. `http://www.linuxfoundation.org/collaborate/workgroups/networking/ifb`.

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

[29] P. Kamp and R. Watson. Jails: Confining the omnipotent root. In *Proc. 2nd Intl. SANE Conference*, pages 99–1. Citeseer, 2000.

[30] M. Lageman and S. Solutions. Solaris Containers—What They Are and How to Use Them. *Sun BluePrints OnLine*, pages 819–2679, 2005.

[31] D. Lin and R. Morris. Dynamics of random early detection. In *Proceedings of the ACM SIGCOMM'97 conference on Applications, technologies, architectures, and protocols for computer communication*, page 137. ACM, 1997.

[32] T. Lindholm and F. Yellin. *Java(tm) Virtual Machine Specification*. Addison-Wesley Professional, 1999.

[33] X. Liu, X. Zhu, P. Padala, Z. Wang, and S. Singhal. Optimal multivariate control for differentiated services on a shared hosting platform. In *Proceedings of the IEEE Conference on Decision and Control*, volume 7. Citeseer, 2007.

[34] J. Luo, J. Pettit, M. Casado, J. Lockwood, and N. McKeown. Prototyping fast, simple, secure switches for ethane. In *Proceedings of the 15th Annual IEEE Symposium on High-Performance Interconnects (HOTI '07)*, pages 73–82, Palo Alto, CA, Agosto de 2007.

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

[36] J. Mudigonda, P. Yalagandula, M. Al-Fares, and J. C. Mogul. 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, Abril de 2010.

[37] R. N. Mysore, A. Pamboris, N. Farrington, N. Huang, S. R. Pardis Miri, V. Subramanya, and A. Vahdat. PortLand: A scalable fault-tolerant layer 2 data center network fabric. In *Proceedings of the ACM SIGCOMM 2009 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, pages 39–50, Barcelona, Spain, Agosto de 2009.

[38] S. Nanda and T. cker Chiueh. A survey of virtualization technologies. Technical report, 2005.

[39] P. Padala, K. Hou, K. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, and A. Merchant. Automated control of multiple virtualized resources. In *Proceedings of the*

fourth ACM european conference on Computer systems, pages 13–26. ACM New York, NY, USA, 2009.

[40] P. Padala, K. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, A. Merchant, and K. Salem. Adaptive control of virtualized resources in utility computing environments. *ACM SIGOPS Operating Systems Review*, 41(3):302, 2007.

[41] A. Parekh and R. Gallager. A generalized processor sharing approach to flow control in integrated services networks: the single-node case. *IEEE/ACM Transactions on Networking (TON)*, 1(3):344–357, 1993.

[42] Performance problems for rackspace cloud, Janeiro de 2010. `http://www.datacenterknowledge.com/archives/2010/01/14/performance-problems-for-rackspace-cloud/`.

[43] L. Peterson and B. Davie. *Computer networks: a systems approach*. Morgan Kaufmann Pub, 2007.

[44] I. Pratt, K. Fraser, S. Hand, C. Limpach, A. Warfield, D. Magenheimer, J. Nakajima, and A. Mallick. Xen 3.0 and the art of virtualization. In *Linux Symposium*. Citeseer, 2005.

[45] Rackspace Cloud, Janeiro de 2010. `http://www.rackspacecloud.com/`.

[46] B. Raghavan, K. V. Vishwanath, S. Ramabhadran, K. Yocum, and A. C. Snoeren. Cloud control with distributed rate limiting. In *Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM '07)*, pages 337–348, Kyoto, Japan, Agosto de 2007.

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

[48] M. Shreedhar and G. Varghese. Efficient fair queueing using deficit round-robin. *IEEE/ACM Transactions on Networking (TON)*, 4(3):385, 1996.

[49] W. Stevens. RFC2001: TCP slow start, congestion avoidance, fast retransmit, and fast recovery algorithms. *RFC Editor United States*, 1997.

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

[51] Token bucket, Fevereiro de 2010. http://en.wikipedia.org/wiki/Token_bucket.

[52] J. Touch. RFC 5556: Transparent interconnection of lots of links (trill): Problem and applicability statement, Maio de 2009.

[53] Visual evidence of Amazon EC2 network issues, Janeiro de 2010. https://www.cloudkick.com/blog/2010/jan/12/visual-ec2-latency/.

[54] J. Watson. Virtualbox: bits and bytes masquerading as machines. *Linux J.*, 2008(166):1, 2008.

[55] A. Whitaker, M. Shaw, and S. D. Gribble. Scale and performance in the denali isolation kernel. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI 2002)*, Boston, MA, Dezembro de 2002.

[56] T. Wood, P. Shenoy, A. Venkataramani, and M. Yousif. Sandpiper: Black-box and gray-box resource management for virtual machines. *Computer Networks*, 2009.