Avaliação de Desempenho de um Proxy HTTP Implementado como Função Virtual de Rede

Rodrigo S. V. Eiras[1,2], **Rodrigo S. Couto**[1], **Marcelo G. Rubinstein**[1] *

[1]Universidade do Estado do Rio de Janeiro - FEN/PEL/DETEL

[2]IesBrazil Consulting and Services - IT Section

{rodrigo.eiras}@iesbrazil.com.br,{rodrigo.couto,rubi}@uerj.br

Resumo. *Virtualização de funções de rede é um paradigma em que serviços de rede são virtualizados sobre hardware genérico. Nessa abordagem, um dos principais desafios é o desempenho quando comparado com o das soluções dedicadas. Este trabalho avalia o desempenho de um proxy HTTP em duas soluções de virtualização, o KVM e o Docker. Os resultados mostram que o Docker possui desempenho superior ao do KVM, apresentando tempos de processamento do proxy mais próximos ao de uma solução sem virtualização. Entretanto, quando um maior isolamento é necessário, o KVM é mais adequado. Nessa linha, este trabalho mostra que a para-virtualização do KVM melhora significativamente o desempenho, mas não o suficiente para superar o Docker.*

Abstract. *Network Functions Virtualization is a paradigm in which network services are virtualized over generic hardware. In this approach, one of the key challenges is the performance when compared to dedicated solutions. This work evaluates the performance of an HTTP proxy in two virtualization solutions, KVM and Docker. Results show that Docker performs better than KVM, presenting proxy processing times close to those of a non-virtualized solution. However, when more isolation is required, KVM is more suitable. In this line, this work shows that the para-virtualization of KVM significantly improves performance, but not enough to overcome Docker.*

1. Introdução

Tradicionalmente, funções de rede como *firewalls*, roteadores e *proxies* são atreladas a dispositivos dedicados. Isso torna o gerenciamento de rede menos flexível e com menor capacidade de expansão. O conceito de virtualização de funções de rede (*Network Functions Virtualization* - NFV) é uma alternativa para solucionar esses problemas, consistindo em implantar as funções de rede em servidores de uso geral. O NFV vem atraindo a atenção da indústria de telecomunicações, em função da possibilidade de redução de custos de operação e de manutenção. Nessa linha, o ETSI (*European Telecommunication Standard Institute*) [ETSI, 2013] vem publicando uma série de especificações que padroniza uma arquitetura NFV.

Uma questão levantada pela indústria de telecomunicações corresponde ao desempenho de uma função virtual de rede (*Virtual Network Function* - VNF)

*Este trabalho foi realizado com recursos da FAPERJ, CNPq e CAPES e dos processos nº 15/24494-8 e nº 15/24490-2, da Fundação de Amparo à Pesquisa do Estado de São Paulo (FAPESP).

quando implantada, por exemplo, em complexas redes de grandes centros de dados [Mijumbi et al., 2016]. Esse tipo de questionamento é feito pois, em um ambiente virtualizado, as instruções computacionais executadas no Sistema Operacional (SO) convidado instalado em uma máquina virtual necessitam ser traduzidas na camada de virtualização, também denominada hipervisor, de forma a garantir o isolamento completo das máquinas virtuais [Couto et al., 2014]. É esperado que aplicações executadas em máquinas virtuais apresentem desempenho inferior quando comparadas com aplicações em ambientes não virtualizados. Além do exposto, cada máquina virtual criada em um ambiente de virtualização tradicional necessita obrigatoriamente ter seu próprio SO, bem como sua configuração de *hardware* virtual dedicado. Isso pode acarretar, sob certas condições, em um desperdício de recursos quando a máquina virtual estiver ociosa. Utilizando a para-virtualização, um ganho de desempenho é notado, uma vez que o SO convidado da máquina virtual consegue, através de modificações, acessar diretamente o *hardware* físico do computador. No entanto, parâmetros da máquina virtual como a quantidade de memória e o número de processadores virtuais ainda precisam ser dimensionados no momento da criação da VNF.

De acordo com [Fernandes et al., 2011], experimentos mostram que para uma mesma configuração de *hardware*, roteadores Linux conseguem atingir uma taxa de encaminhamento de pacotes em torno de 1,2 M pacotes/s. Já em um roteador virtual Linux virtualizado com o Xen [Barham et al., 2003], essa taxa atinge aproximadamente 0,2 M pacotes/s. Dessa forma, é possível concluir que aplicações que demandam uma alta taxa de encaminhamento de pacotes enfrentam problemas de desempenho quando utilizadas em ambientes de virtualização tradicionais. Isso mostra o desafio de implementar funções de rede como VNFs. Uma alternativa a esse problema é a adoção da virtualização leve, também denominada conteinerização. Nesse tipo de virtualização, o isolamento é aplicado apenas no nível de processo, ao invés de a todo o SO. Uma vez que contêineres não sofrem do *overhead* imposto pelos hipervisores, é esperada uma melhora no desempenho das aplicações virtualizadas. Dado o exposto, é importante analisar quando uma função de rede pode ser virtualizada e qual o tipo de solução de virtualização adequado. A utilização de contêineres pode ser uma alternativa interessante ao criar VNFs, pois devido ao formato da arquitetura de funcionamento dos mesmos e somado com a ausência de uma camada de hipervisor, o desempenho de uma aplicação utilizando contêiner pode estar bem próximo a de um sistema Linux nativo.

Uma das funções que podem ser implementadas como VNFs é o *proxy*. O *proxy* HTTP (*HyperText Transfer Protocol*) é responsável por responder requisições HTTP em nome de servidores. Com isso, é possível, por exemplo, filtrar requisições, permitindo o acesso a somente determinadas páginas de servidores. Além disso, pode-se reduzir o tempo de resposta a uma requisição e a ocupação de enlaces de acesso quando o *proxy* possui também um cache associado. Ainda de acordo com [Squid, 2017], serviços de *proxy* são bastante sensíveis à utilização do processador (CPU) e isso pode se tornar um empecilho quando esse tipo de função de rede necessita ser virtualizada.

Em [Eiras et al., 2016] é realizada uma avaliação de desempenho de um único *proxy* HTTP implementado como uma VNF em duas plataformas de virtualização caracterizadas como *softwares* livres: o KVM (*Kernel-based Virtual Machine*) [KVM, 2017], por padrão, uma plataforma de virtualização completa, e o Docker [Docker, 2017b], que im-

plementa virtualização leve através de contêineres. Os resultados mostram que a solução com um contêiner Docker consegue ter um desempenho próximo ao da solução com Linux nativo, se mostrando uma boa opção à solução utilizando o KVM.

Uma vez que os resultados indicam a viabilidade do uso de um contêiner para implementação de um *proxy* HTTP como VNF, este trabalho estende o trabalho anterior avaliando o desempenho de diversas instâncias de *proxy* sendo executados ao mesmo tempo. Esse tipo de avaliação é importante, pois na prática a quantidade de *proxies* deve variar em função da demanda (por exemplo, em função do número de requisições recebidas). Pode-se observar que o uso de duas instâncias de *proxy* melhora o desempenho em relação ao cenário com apenas um *proxy*. Nesse caso, contêineres do tipo Docker se aproximam ainda mais do Linux nativo em termos de desempenho. Os resultados também mostram que para mais de dois *proxies*, no cenário considerado, a melhora no desempenho não é significativa. Além disso, este trabalho avalia o desempenho de máquinas para-virtualizadas KVM. O objetivo dessa avaliação é verificar se a para-virtualização é uma alternativa ao Docker, quando há uma preocupação maior com flexibilidade e isolamento. Os resultados mostram também que, em situações em que uma queda de desempenho seja tolerável, o uso de uma máquina virtual KVM para-virtualizada pode ser considerado, com um desempenho aproximadamente 50% superior em relação ao KVM com virtualização completa.

Este trabalho está organizado da seguinte forma. A Seção 2 apresenta os trabalhos relacionados. A Seção 3 descreve as principais características dos tipos de virtualizações utilizados neste trabalho. Na Seção 4 é apresentada a avaliação de desempenho realizada. Por último, as conclusões e os trabalhos futuros são apresentados na Seção 5.

2. Trabalhos Relacionados

Diversos trabalhos avaliam o desempenho de soluções para ambientes de virtualização. [Kim et al., 2010] avaliam de que forma transferências de grandes volumes de dados impactam o desempenho de um sistema de *proxy* Web com cache. São avaliadas métricas como latência e vazão em uma rede CAN (*Campus Area Network*), focando somente em casos que podem gerar um mau comportamento do cache ou falhas de serviços. Entretanto, esse trabalho não avalia o desempenho do sistema de *proxy* com cache em plataformas que suportam NFV.

[Dua et al., 2014] apresentam uma análise teórica de soluções baseadas em hipervisores e contêineres. São comparadas características como isolamento, armazenamento de dados e comunicação de rede. O foco principal está nos contêineres e em sua capacidade de hospedar aplicações. A análise é relevante, porém métricas de desempenho com foco em NFV não foram avaliadas.

[Yang e Lan, 2015] propõem um modelo de avaliação de desempenho de máquinas virtuais no KVM. São avaliados parâmetros que podem impactar o desempenho do ambiente virtualizado, tais como velocidade de processamento, operações com ponto flutuante e taxa de cópia de arquivos. Dos resultados, um índice é gerado para comparar as soluções em uma escala de eficiência em cenários envolvendo servidores *web* e não é considerado o uso de contêineres.

[Felter et al., 2015] avaliam o desempenho do MySQL, uma aplicação de banco de dados, sendo executada no KVM e no Docker. Os resultados mostram uma melhora no

desempenho do serviço quando se utiliza virtualização por contêiner. Contudo, aplicações de banco de dados não são adequadas para avaliar o desempenho de tráfego de rede entre plataformas virtuais e a aplicabilidade dos contêineres em serviços de NFV.

[Bondan et al., 2016] analisam três soluções de virtualização, o ClickOS [Martins et al., 2014], o CoreOS [Docker, 2017a] e o OSv [Kivity et al., 2014], com foco em monitoramento de recursos. É avaliado o tempo necessário para executar operações do tipo criar e inicializar uma nova VNF em contêineres e em máquinas virtuais, bem como uso de memória. Não é avaliado o desempenho das soluções de contêiner para aplicações de *proxy* HTTP, uma vez que os autores focam em métricas de monitoramento de VNFs.

[Heideker e Kamienski, 2016] apresentam o NFV e o SDN (*Software Defined Network*) como possíveis soluções a desafios como flexibilidade e qualidade de serviço em tecnologias como Internet das Coisas (*Internet of Things* - IoT) e Cidades Inteligentes (*Smart Cities*). O NFV é empregado para solucionar o problema de elasticidade no acesso público à Internet em grandes cidades. Mais especificamente, é apresentada uma solução para o gerenciamento de NAT (*Network Address Translator*) em um ambiente com praças digitais. As avaliações são realizadas utilizando-se o KVM e o Xen como plataformas de virtualização tradicionais e contêineres LXC como plataforma de virtualização leve. Nos testes executados, o LXC consegue oferecer melhores resultados em relação ao KVM, como por exemplo na vazão média, cenário em que o LXC tem um desempenho 80% superior. O trabalho está relacionado à infraestrutura para NFV, no entanto não considera o uso de *proxies* HTTP na arquitetura proposta e não utiliza o Docker como tecnologia de contêiner.

Apesar de NFV ser um assunto extremamente explorado pela academia e pela indústria de telecomunicações, a literatura é carente de trabalhos que exploram a utilização de *proxies* HTTP implementados como uma função de rede virtualizada. Dessa forma, este trabalho avalia o desempenho de duas soluções de virtualização, o KVM e o Docker, quando utilizadas em um *proxy* HTTP.

3. Soluções de Virtualização

Soluções de virtualização multiplexam o acesso ao *hardware* físico, permitindo a criação de múltiplas instâncias virtuais em uma mesma máquina. Como o NFV também se baseia na multiplexação do *hardware* de rede, VNFs podem ser executadas sobre soluções de virtualização originalmente concebidas para centros de dados. Neste trabalho, essas soluções são classificadas em virtualização completa, para-virtualização e virtualização leve ou conteinerização.

3.1. Virtualização Tradicional

Soluções de virtualização tradicionais utilizam hipervisores para multiplexar o acesso ao *hardware* pelas máquinas virtuais. Para tal, implementam uma camada de *software* denominada hipervisor, que se encontra entre o *hardware* da máquina física e as máquinas virtuais. O hipervisor controla o acesso ao *hardware* e é responsável por entregar ao SO hospedeiro a abstração das máquinas virtuais, provendo assim isolamento entre essas. Dessa forma, máquinas virtuais podem implementar diferentes SOs, denominados SOs convidados, e consequentemente executar diferentes aplicações. Pode-se afirmar que

cada máquina virtual possui sua própria abstração de *hardware* virtual e seu próprio SO, que é responsável por controlar os dispositivos necessários ao seu funcionamento; por exemplo, processadores, memórias, discos rígidos e outros.

Dependendo das necessidades de um projeto de virtualização, a implementação da virtualização pode ser total, denominada virtualização completa, ou parcial, denominada para-virtualização. Esses dois tipos se diferem em relação às chamadas que passam pelo hipervisor, como detalhado adiante. É importante ressaltar que, independentemente da solução de virtualização a ser utilizada, o ambiente deve ser cuidadosamente estudado e corretamente dimensionado, uma vez que essas soluções aumentam a complexidade e alteram a dinâmica de funcionamento da infraestrutura utilizada.

3.1.1. Virtualização Completa

Na virtualização completa, todas as instruções das máquinas virtuais são interceptadas pelo hipervisor antes de chegarem ao *hardware*. Assim, não há necessidade de se alterar qualquer parte do SO para ser executado em uma máquina virtual [Barham et al., 2003]. Existem diversas soluções de virtualização completa disponíveis atualmente, tais como VMware ESXi [VMware, 2012], Citrix Xen Server [Citrix, 2017], Microsoft Hyper-V [Microsoft, 2017], Oracle Virtualbox [Oracle, 2017], entre outras. Neste trabalho, utiliza-se o hipervisor KVM [KVM, 2017], por ser um *software* livre e em função de sua integração com o *kernel* do Linux.

A principal desvantagem do uso da virtualização completa em NFV é o gargalo de desempenho imposto às máquinas virtuais pelo hipervisor. Uma vez que todas as operações de comunicação da máquina virtual necessitam primeiramente passar pela camada de hipervisor antes de chegarem ao *hardware*, as aplicações executadas em um ambiente virtualizado podem ter uma queda de desempenho quando comparadas com aplicações que não usam virtualização [Fernandes et al., 2011].

3.1.2. Para-Virtualização

O conceito de para-virtualização, também denominada virtualização parcial, é similar ao princípio da virtualização completa. Um hipervisor é instalado sobre o *hardware* de uma máquina física, seja um computador pessoal ou um servidor comum, e então máquinas virtuais convidadas podem ser implementadas sobre esse hipervisor. A diferença está no fato de que o SO da máquina virtual convidada está ciente de sua virtualização. Assim, pode realizar chamadas diretamente ao *hardware*, melhorando o desempenho. Os SOs convidados requerem extensões a serem aplicadas no momento da implementação da virtualização para que possam fazer chamadas de sistemas diretamente ao *hardware*. As alterações no SO são implementadas como uma alternativa à tradução, pelo hipervisor, de funções complexas de serem virtualizadas e que geram algum *overhead* no ambiente quando são executadas. Um exemplo disso são as instruções privilegiadas de nível (anel) 0 (zero), que para acessar certos setores da memória RAM ou realizar chamadas de *drivers* de dispositivos de E/S (Entrada/Saída), são então executados diretamente no *hardware* [Babu et al., 2014]. No ambiente implementado para

este trabalho, interfaces de rede para-virtualizadas do tipo VirtIO [VirtIO, 2017] são implementadas. O VirtIO é um *framework* desenvolvido para acelerar instruções de E/S em um ambiente virtualizado com KVM. Em poucas palavras, o VirtIO fornece uma série de *drivers* e filas compartilhadas entre o SO convidado e o SO hospedeiro com a finalidade de diminuir o *overhead* imposto pelo hipervisor [Nakajima et al., 2015].

Grande parte dos ambientes nos quais se encontram soluções para-virtualizadas se baseia no Linux. Somado ao fato de esse trabalho ter como motivação a utilização de *software* livre e a utilização do Linux como sistema operacional, a para-virtualização se enquadra nas premissas previamente estabelecidas como uma possível solução de virtualização para NFV. Foi avaliado o desempenho do KVM considerando para-virtualização.

3.2. Virtualização Leve ou Conteinerização

A virtualização leve, também denominada conteinerização, oferece um nível diferente de virtualização e isolamento quando comparada à virtualização tradicional. Diferentemente da virtualização completa e da para-virtualização, contêineres implementam o isolamento entre as aplicações a nível de processos no SO base, evitando assim o *overhead* imposto pela abstração de *hardware* do hipervisor nas máquinas virtuais. Contêineres compartilham o mesmo SO instalado na máquina física e uma ou mais aplicações podem ser executadas dentro de um ou mais contêineres.

Nas soluções baseadas em contêineres, o fato de o *kernel* do SO hospedeiro ser compartilhado provê vantagens como, por exemplo, uma capacidade de provisionamento de grande quantidade de instâncias de uma aplicação qualquer em um curto espaço de tempo. Isso ocorre pois as instâncias não necessitam de imagens de disco rígido virtual, ao contrário da virtualização tradicional. Entretanto, devido ao compartilhamento do SO e de suas bibliotecas entre os contêineres, não é possível executar diferentes SOs em uma mesma máquina física. Outra desvantagem ao implementar a tecnologia de contêineres é o método de isolamento provido por essas soluções. Uma vez que o *kernel* é exposto aos contêineres, podem ocorrer problemas de segurança que afetam todo o ambiente no qual a solução de contêiner está implementada [Bui, 2015].

Neste trabalho, para avaliar o desempenho de soluções de contêineres, utiliza-se o Docker [Docker, 2017b]. O Docker é uma solução de virtualização baseada em contêineres que possui grande aceitação e consequente adoção por parte de desenvolvedores para fornecimento de microsserviços.

4. Avaliação de Desempenho

Esta seção avalia o desempenho de *proxies* HTTP em contêineres Docker, em máquinas virtuais KVM e em uma máquina física. O ambiente de testes inclui uma máquina com clientes HTTP e outra máquina com um servidor HTTP. Essas duas máquinas estão interconectadas através de uma máquina intermediária, que possui os *proxies*. Os clientes HTTP são executados como contêineres. O servidor HTTP é executado diretamente na máquina física com Linux nativo. O desempenho dos contêineres clientes e do servidor HTTP não influenciam as medidas realizadas, que dependem apenas dos *proxies* da máquina intermediária.

Um *proxy* pode ser uma máquina física no caso do Linux nativo, uma máquina virtual para o KVM ou um contêiner para o Docker. Quando são necessárias duas ou mais instâncias de *proxy* no Linux nativo, são criados *sockets* adicionais que utilizam portas diferentes. Já no caso do contêiner e da máquina virtual, um novo contêiner ou uma nova máquina virtual com diferentes endereços IPs são criados quando mais instâncias de *proxy* são necessárias.

Em relação às máquinas virtuais KVM, além do padrão que utiliza virtualização completa, os experimentos utilizam interfaces de rede virtuais implementadas com *drivers VirtIO*. O Docker também é configurado de duas formas nos experimentos. A primeira é a padrão, denominada neste trabalho como Docker-NAT, na qual o encaminhamento de requisições para o contêiner é realizado com NAT. A segunda, denominada Docker-Roteado, utiliza regras de roteamento estático para encaminhar as requisições para o contêiner.

Nos cenários avaliados, as três máquinas possuem configurações de *hardware* idênticas, cada uma com um processador quad-core Intel Xeon 3,2 GHz, 16 GB de memória RAM e quatro interfaces de rede de 1 Gb/s. Para evitar interferências de agentes externos, as máquinas são interconectadas diretamente por cabos *crossover*. O SO utilizado tanto nas máquinas físicas quanto nas máquinas virtuais e contêineres é o Ubuntu Linux 14.04 LTS de 64 bits. A geração de requisições HTTP bem como a extração dos tempos de processamento dessas requisições são realizadas pela ferramenta Apache JMeter [Apache, 2017] versão 3.1. O tempo de processamento de uma requisição é definido como o tempo entre o envio da requisição e o recebimento de sua resposta. Os servidores HTTP utilizam o Apache 2, enquanto os *proxies* em qualquer uma das configurações utilizam a versão 3 do *proxy* Squid. Existem diversas plataformas no mercado que podem prover sistemas de cache a uma arquitetura corporativa. No entanto, optou-se por escolher o Squid 3 por ser de código aberto e também ser um dos mais populares servidores de *proxy* HTTP entre analistas e engenheiros de telecomunicações. O tamanho do objeto a ser transferido entre o servidor HTTP e o cliente é de 1024 kBytes, uma vez que experimentos com tamanhos menores de objetos levam a conclusões similares. É importante frisar que as máquinas virtuais no KVM são configuradas com 4 GB de memória RAM cada e um processador virtual quad-core. Os contêineres Docker também são configurados respeitando-se a mesma configuração do KVM; ou seja, utilizando-se quatro núcleos de processador e com limite de 4 GB de memória RAM cada.

Os experimentos são divididos em duas partes. A primeira foca a avaliação do tempo médio necessário para processar requisições HTTP que partem de 10 clientes para o servidor alvo, passando por uma ou duas instâncias de *proxy*. São realizadas análises sem e com caches habilitados nos *proxies*. Em seguida, estendendo os resultados da primeira parte, avalia-se o quão satisfatória é a distribuição da carga de requisições processadas pelo serviço de *proxy* em duas ou mais instâncias de contêineres.

Em uma amostra de experimento, cada cliente envia simultaneamente 1000 requisições HTTP para o servidor e o tempo total é dividido por 1000, obtendo-se assim o tempo médio por requisição. Cada experimento é realizado 10 vezes e as médias dos tempos médios são apresentadas juntamente com os intervalos de confiança de 95%. Entretanto, na maioria dos resultados, os intervalos de confiança são imperceptíveis. De forma a avaliar o desempenho das soluções em um ambiente com alta carga, um aumento

de carga é induzido na máquina física que executa as instâncias de *proxy*. É utilizada a ferramenta Stress [Stress, 2014] para criar processos do tipo *fork* que consomem recursos do *hardware* e elevam a utilização do processador. Para tal, o comando *stress –cpu 12 –timeout 48h* é utilizado, no qual *–cpu* indica a quantidade de processos do tipo *fork* e *–timeout* indica a duração do aumento de carga.

4.1. Tempo Médio de Processamento com Caches Desabilitados

No primeiro cenário, avalia-se o tempo médio de processamento das requisições HTTP enviadas por 10 clientes utilizando uma ou duas instâncias de *proxy*. Quando virtuais, as instâncias são configuradas no KVM ou no Docker. No caso do Linux nativo, dois *sockets* do *proxy* são inicializados com portas diferentes. A ideia ao dividir as requisições entre duas instâncias de *proxy* é verificar se existe alguma melhoria considerável de tempo nesse tipo de implementação, uma vez que requisições poderão ser processadas em paralelo. Para esse cenário, o sistema de cache do Squid 3 está desabilitado. Implementações de *proxy* HTTP com sistema de cache desabilitado são comuns em cenários nos quais somente é necessário o uso do *proxy* para filtragem e restrição de navegação na web.

A Figura 1 apresenta o tempo médio por requisição com uma ou duas instâncias de *proxy*, sem caches. É possível notar que o KVM, quando utilizando técnicas de paravirtualização, consegue melhorar o desempenho em relação à virtualização completa, mas não o suficiente para superar o desempenho do contêiner. Assim, o Docker tem um desempenho mais próximo ao do Linux nativo, independentemente se encaminha pacotes por roteamento ou NAT para o contêiner. O Docker se beneficia da inexistência de um hipervisor, que insere um *overhead* significativo na comunicação entre o *hardware* e a máquina virtual. No entanto, contêineres configurados com NAT ainda têm um *overhead* ligeiramente maior que contêineres que utilizam roteamento.

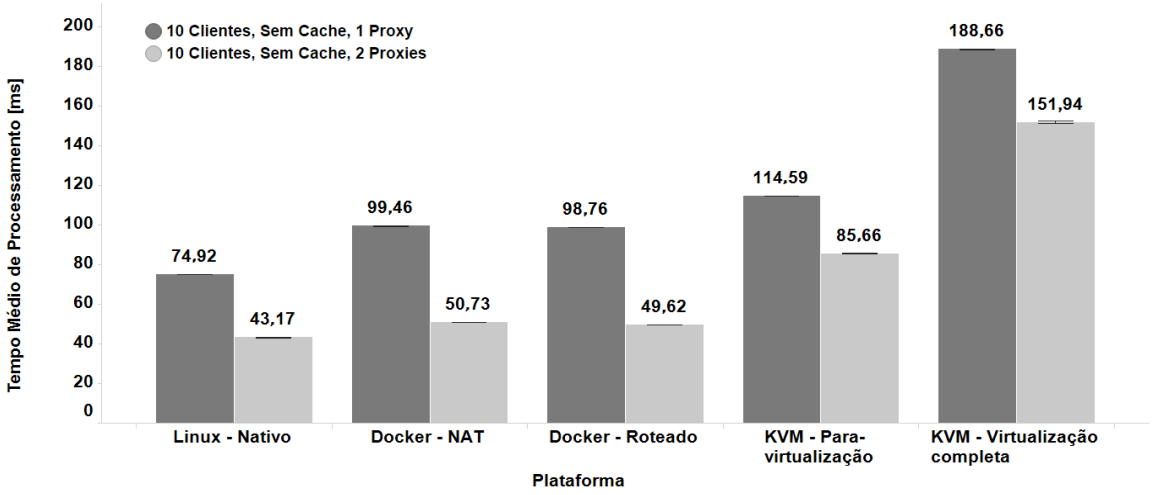

Figura 1. Tempo médio de processamento por requisição enviada por 10 clientes com uma ou duas instâncias de *proxy* e cache desabilitado.

No modo NAT, o Docker utiliza uma interface virtual anexada (*bridge*) à interface física e, nesse caso, cada contêiner recebe um endereço de rede privado para permitir sua comunicação com a rede externa. É importante salientar que o uso do NAT pode oferecer uma camada de segurança adicional para todas as conexões que entram no contêiner

ao custo de um pequeno *overhead*, não permitindo que os contêineres fiquem expostos ao mundo externo. A Tabela 1 mostra uma comparação de utilização de processador e memória para as duas formas de implementação de contêineres do Docker servindo como VNF de *proxy*. Para permitir que a utilização de processador e memória sejam aferidos de forma correta, a imposição de estresse presente nos experimentos anteriores foi removida. Especificamente para esse experimento de medida de processamento e memória, utiliza-se apenas um cliente fazendo requisição para uma VNF. É possível perceber que os contêineres implementados com NAT possuem um *overhead* de CPU maior em relação à versão roteada. A versão roteada, por sua vez, é mais sensível ao uso de memória, pois precisa armazenar os endereços de destino das conexões para os contêineres. Quando as

Tabela 1. Utilização de memória e processador nas versões de VNFs utilizando NAT e Roteamento no Docker.

Plataforma	Memória	Processador (CPU)
Docker - Roteado	3,75% $\pm$ 0,001	40,75% $\pm$ 0,02
Docker - NAT	1% $\pm$ 0,001	42,67% $\pm$ 0,02

requisições são divididas em mais de um serviço de *proxy* para processamento, a Figura 1 mostra que os tempos diminuem consideravelmente. Essa queda é ainda maior quando contêineres são utilizados, nos quais a adição de mais um *proxy* leva a uma queda de aproximadamente 50% no tempo de processamento. O comportamento é esperado uma vez que requisições podem ser processadas em paralelo. Outra discussão que pode ser considerada nesse ponto está relacionada ao provisionamento de mais instâncias de *proxy* quando necessárias. Soluções baseadas em contêineres podem ser instanciadas rapidamente para, por exemplo, suprir uma sobrecarga momentânea [Bondan et al., 2016]. Isso é possível pois, ao contrário da virtualização tradicional, não existe alocação prévia de recursos, apenas limites máximos de uso de memória e CPU.

4.2. Tempo Médio de Processamento com Caches Habilitados

Em um segundo cenário, avalia-se o tempo médio de processamento das requisições HTTP com o sistema de cache do Squid 3 habilitado. Em um cenário com o cache habilitado, a primeira requisição de um determinado objeto é obtida do servidor HTTP. A partir disso, o Squid 3 armazena localmente uma cópia do objeto, que é obtida diretamente do cache para as requisições subsequentes. Para habilitar o sistema de cache no Squid 3, alguns requisitos devem ser considerados para o funcionamento correto. Dois parâmetros são ajustados nas configurações; o primeiro denominado *maximum object size* é relacionado ao tamanho máximo do objeto a ser armazenado no cache. A documentação do Squid 3 não recomenda que esse valor seja maior que 1024 kBytes para evitar problemas de desempenho. O segundo parâmetro ajustado é o tamanho do espaço em memória RAM (*total size of memory*) a ser dedicado ao serviço de *proxy*. Utiliza-se o valor padrão indicado no arquivo de configuração, que é de 260 MBytes. Todos os demais parâmetros são mantidos com valores padrão, e nenhuma outra espécie de cache é utilizada, seja no lado do servidor HTTP ou do cliente. Espera-se que nesse cenário os tempos de processamento naturalmente sejam reduzidos. O objetivo desta seção é verificar se o comportamento das soluções de virtualização se mantém o mesmo do cenário com cache desabilitado.

A opção de se avaliar uma instância de *proxy* com sistema de cache habilitado historicamente alinha com a necessidade de preservar banda de navegação para clientes espalhados na rede. Com o crescimento da tecnologia de redes ao longo do tempo, banda de transmissão é uma menor preocupação para redes WAN e LAN ligadas diretamente a um *backbone*, através de fibra óptica ou cabo de par metálico. No entanto, com o advento de dispositivos que utilizam cada vez mais conexões WAN sem fio, por exemplo 3G ou 4G, a necessidade da utilização de cache volta a ser considerada pela indústria. Assim, é possível economizar banda de navegação e reduzir a saturação de equipamentos e rotas de tráfego de rede [Technology, 2013].

A Figura 2 apresenta o tempo médio de processamento das requisições HTTP com uma ou duas instâncias de *proxy* e caches habilitados. Assim como no experimento anterior, é possível perceber que contêineres Docker possuem desempenho mais próximo ao do Linux nativo, e essa proximidade aumenta para duas instâncias de *proxy*. Da mesma forma que o observado nos resultados sem cache, o desempenho do KVM para-virtualizado e do Docker se aproximam. Apesar dos melhores resultados do Docker é importante salientar que o hipervisor, nas soluções de virtualização tradicional, separa um espaço de memória RAM exclusivo para a máquina virtual. Como sistemas de cache utilizam memória RAM intensamente para armazenar objetos recentemente requisitados por clientes, dedicar espaço em RAM melhora o isolamento da solução.

É importante frisar que, independentemente da tecnologia a ser utilizada, o uso de caches pode melhorar consideravelmente o desempenho de uma solução de *proxy* HTTP como função virtual de rede.

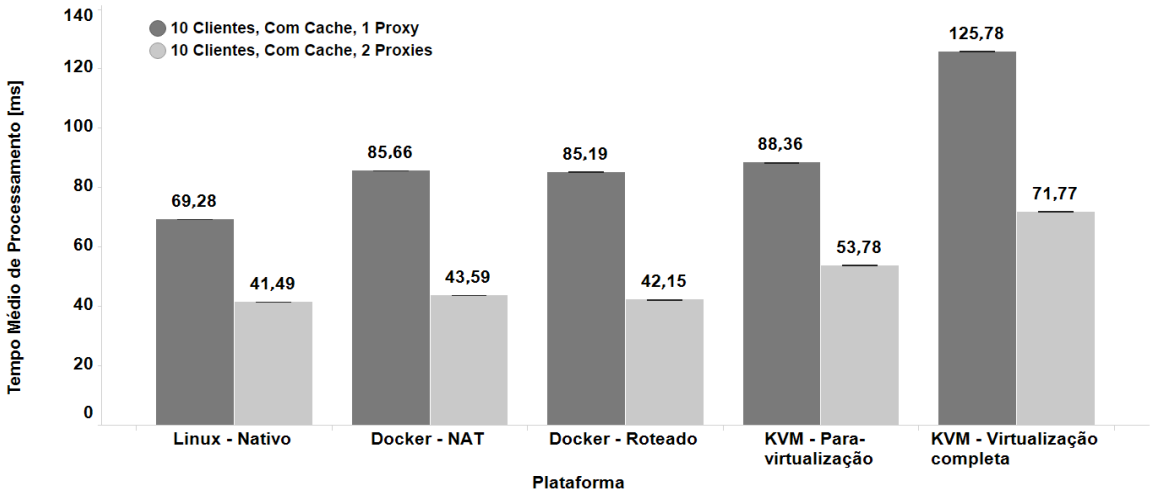

Figura 2. Tempo médio de processamento por requisição enviada por 10 clientes com uma ou duas instâncias de *proxy* e cache habilitado.

4.3. Análise do Balanceamento de Carga do Docker

Nas Seções 4.1 e 4.2 comparam-se as diferentes soluções de virtualização na utilização de *proxies* em NFV. Os resultados mostram que contêineres Docker possuem desempenho mais próximo ao do Linux Nativo. Além disso, a utilização de dois *proxies* para balancear a carga melhora o desempenho, aproximando o Docker ainda mais do Linux nativo. Uma vez que o Docker obteve o melhor desempenho entre as tecnologias de

virtualização, esta seção visa analisar em mais detalhes sua capacidade de balanceamento de carga. Para tal, foram criados 64 contêineres clientes para requisição de arquivos de 1024 KBytes por HTTP. Esse número de clientes foi escolhido de forma a possibilitar variar o número de *proxies* em potência de dois e garantir que esses *proxies* recebam quantidades iguais de clientes. Os *proxies* são configurados com cache, já que essa é a configuração com melhores resultados nos experimentos anteriores.

Cada cliente executa uma instância do Apache JMeter para enviar as requisições. Na máquina que controla as instâncias de *proxy* virtuais, são criados até 32 contêineres executando o Squid 3. Essa quantidade está próxima ao número de contêineres Docker suportados pela máquina utilizada. A partir de 38 instâncias, os contêineres começam a recusar conexões com mensagem de "sem rota para o destino". Com os contêineres criados, ativam-se os *proxies* de acordo com o número desejado. Assim, o tráfego dos 32 clientes é dividido de forma igual entre os *proxies*. Por exemplo, para quatro *proxies*, ativam-se quatro contêineres, que recebem tráfego de 16 clientes cada um.

A Figura 3 mostra os resultados do tempo de resposta médio para os 64 clientes de acordo com o número de *proxies*. Note que o tempo de resposta aumenta consideravelmente em relação aos resultados obtidos anteriormente, dado o aumento do número de clientes. É possível também notar que o balanceamento com duas instâncias provoca uma melhora perceptível no tempo de processamento, de aproximadamente 70 ms, em relação à utilização de apenas um *proxy*. Esse comportamento também é observado nos experimentos das Seções 4.1 e 4.2. Assim como visto nesses experimentos, a Figura 3 mostra que a versão com NAT habilitado e 1 (um) *proxy* possui um *overhead* que culmina em tempos ligeiramente piores em relação à versão roteada. Isso acontece pois o NAT é sensível ao uso de CPU [Tsetse et al., 2012]. A versão roteada, por sua vez, é sensível ao uso de memória, pois precisa armazenar os endereços de destino das conexões para os contêineres. Como a tabela de roteamento é a mesma tanto para os contêineres como para o SO hospedeiro, o uso de memória tende a crescer sensivelmente à medida que novas instâncias são adicionadas ao experimento. Quando as requisições começam a ser distribuídas em mais contêineres, o uso de memória acaba aumentando, por vezes até mesmo fazendo o uso de memória de troca. Assim, quando existem *proxies* paralelos e muitos clientes, o Docker com roteamento passa a ter desempenho ligeiramente inferior ao Docker com NAT.

Para mais de duas instâncias em paralelo, não é percebida uma melhora de desempenho. Além disso, é possível notar que a partir de oito nós em paralelo, os tempos começam a aumentar. Isso ocorre devido a problemas de isolamento do Docker, no qual os processadores e memórias são compartilhados com todo o sistema e podem interferir no desempenho dos contêineres em geral [Combe et al., 2016].

5. Conclusão

NFV é um assunto que vem chamando a atenção da indústria de telecomunicações devido à sua proposta de flexibilização e redução de custos nas implementações de funções de redes. Para substituir com eficiência equipamentos dedicados por soluções virtuais, o desempenho da função virtual de rede é um fator importante e que deve ser cuidadosamente analisado. Nos experimentos mostrados neste trabalho é possível concluir que contêineres criados na plataforma Docker conseguem ter um desempenho de

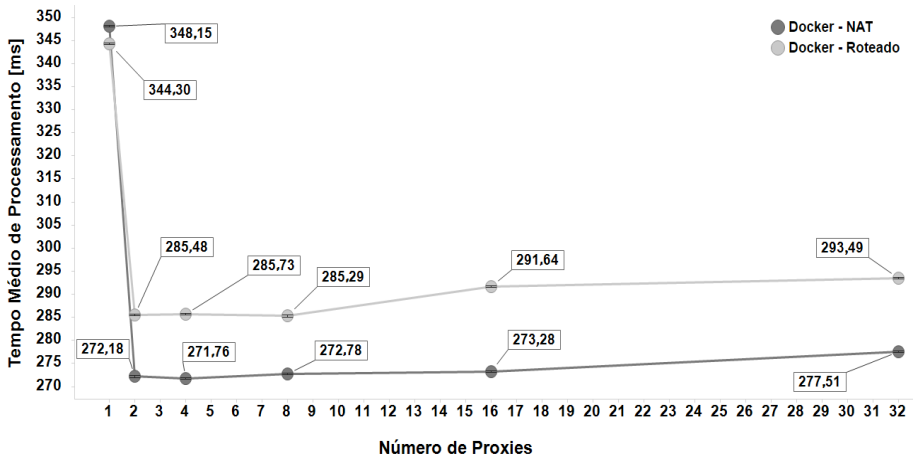

Figura 3. Tempo médio de processamento por requisição enviada por 64 clientes para no máximo 32 instâncias de *proxy* e cache habilitado.

rede mais próximo ao Linux nativo quando aplicados em *proxies* HTTP.

Apesar do desempenho superior do Docker, soluções de virtualização tradicionais, como o KVM, oferecem maior isolamento às aplicações NFV. Neste trabalho avaliou-se o *proxy* HTTP em uma solução de para-virtualização do KVM, denominada VirtIO. Os resultados mostram que essa solução melhora significativamente o tempo de processamento do *proxy*, mas não o suficiente para superar o Docker. Assim, em situações nas quais a virtualização tradicional é necessária, é possível utilizar técnicas de para-virtualização para melhorar o desempenho. Como visto em [Rasmusson e Corcoran, 2014], é importante salientar que a interface VirtIO é dependente de o quanto o *hardware* da máquina física é dedicado às máquinas virtuais; ou seja, quanto mais recursos forem acessados diretamente pelas máquinas virtuais, melhor será o desempenho.

Este trabalho também analisou o quanto o balanceamento de carga pode melhorar no desempenho dos *proxies* virtuais. Os resultados mostram que, em todas as soluções de virtualização, instanciar duas instâncias de *proxy* melhora o desempenho em relação ao cenário com apenas um *proxy*. Nesse caso, contêineres do tipo Docker se aproximam ainda mais do Linux nativo em termos de desempenho. Por fim, analisou-se a capacidade de balanceamento do Docker de acordo com o aumento do número de *proxies* em até 32 instâncias. Os resultados mostraram que para mais de dois *proxies*, no cenário considerado, a melhora no desempenho não é significativa. Além disso, aumentar o número de contêineres aumenta o consumo de memória, podendo gerar queda de desempenho.

Como trabalhos futuros, é interessante avaliar a escalabilidade das tecnologias de conteinerização quando implementadas em NFV, inclusive considerando a utilização de soluções que proveem gerenciamento e escalabilidade para contêineres, como o Kubernetes [Kubernetes, 2017]. É importante também estender este trabalho e avaliar como o isolamento provido pelo Docker pode afetar a segurança de aplicações NFV. Uma vez que apenas o Docker como solução de contêiner foi avaliada para utilização como *proxy* HTTP, outras soluções de contêineres podem ser avaliadas para complementar os resultados mostrados neste trabalho.

Referências

Apache (2017). Jmeter 3.1. http://jmeter.apache.org - Acessado em dezembro de 2017.

Babu, A., M, H., Martin, J. P., Cherian, S. e Sastri, Y. (2014). System performance evaluation of para virtualization, container virtualization and full virtualization using Xen, OpenVZ and XenServer. Em *International Conference on Advances in Computing and Communications*, p. 247–250.

Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I. e Warfield, A. (2003). Xen and the art of virtualization. Em *ACM Symposium on Operating Systems Principles (SOSP)*, p. 164–177.

Bondan, L., dos Santos, C. R. P. e Granville, L. Z. (2016). Comparing virtualization solutions for NFV deployment: A network management perspective. Em *IEEE Symposium on Computers and Communication (ISCC)*, p. 669–674.

Bui, T. (2015). Analysis of Docker security. Relatório técnico. http://arxiv.org/abs/1501.02967.

Citrix, X. (2017). Citrix XenServer. https://www.citrix.com.br/products/xenserver - Acessado em dezembro de 2017.

Combe, T., Martin, A. e Pietro, R. D. (2016). To Docker or not to Docker: A security perspective. *IEEE Cloud Computing*, 3(5):54–62.

Couto, R. S., Campista, M. E. M. e Costa, L. H. M. K. (2014). Network resource control for Xen-based virtualized software routers. *Computer Networks*, 64:71–88.

Docker (2017a). CoreOS: Open source projects for Linux containers. https://coreos.com - Acessado em dezembro de 2017.

Docker (2017b). Docker - build, ship and run any app, anywhere. https://www.docker.com - Acessado em dezembro de 2017.

Dua, R., Raja, A. R. e Kakadia, D. (2014). Virtualization vs containerization to support PaaS. Em *IEEE International Conference on Cloud Engineering*, p. 610–614.

Eiras, R. S. V., Couto, R. S. e Rubinstein, M. G. (2016). Performance evaluation of a virtualized HTTP proxy using KVM and Docker. Em *International Conference on the Network of the Future (NOF)*, p. 1–5.

ETSI, N. (2013). Network functions virtualisation (NFV) architectural framework. *ETSI GS NFV*, 2(2):V1.

Felter, W., Ferreira, A., Rajamony, R. e Rubio, J. (2015). An updated performance comparison of virtual machines and Linux containers. Relatório técnico. IBM Research Report - RC25482 (AUS1407-001).

Fernandes, N. C., Moreira, M. D. D., Moraes, I. M., Ferraz, L. H. G., Couto, R. S., Carvalho, H. E. T., Campista, M. E. M., Costa, L. H. M. K. e Duarte, O. C. M. B. (2011). Virtual networks: Isolation, performance, and trends. *Annals of Telecommunications*, 66(5-6):339–355.

Heideker, A. e Kamienski, C. (2016). Gerenciamento flexível de infraestrutura de acesso público à Internet com NFV. Em *Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos*, p. 937–950.

Kim, H.-C., Lee, D., Chon, K., Jang, B., Kwon, T. e Choi, Y. (2010). Performance impact of large file transfer on web proxy caching: A case study in a high bandwidth campus network environment. *Journal of Communications and Networks*, 52:52–66.

Kivity, A., Laor, D., Costa, G., Enberg, P., Har'El, N., Marti, D. e Zolotarov, V. (2014). OSv — optimizing the operating system for virtual machines. Em *USENIX Annual Technical Conference (USENIX ATC 14)*, p. 61–72.

Kubernetes (2017). Kubernetes. https://kubernetes.io - Acessado em dezembro de 2017.

KVM (2017). Kernel-based virtual machine. https://openvirtualizationalliance.org/what-kvm/overview - Acessado em dezembro de 2017.

Martins, J., Ahmed, M., Raiciu, C., Olteanu, V., e Honda, M. (2014). ClickOS and the art of network function virtualization. Em *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, p. 459–473.

Microsoft (2017). Microsoft Hyper-V. https://msdn.microsoft.com/pt-br/library/hh831531(v=ws.11).aspx - Acessado em dezembro de 2017.

Mijumbi, R., Serrat, J., Gorricho, J. L., Bouten, N., Turck, F. D. e Boutaba, R. (2016). Network function virtualization: State-of-the-art and research challenges. 18(1):236–262.

Nakajima, Y., Masutani, H. e Takahashi, H. (2015). High-performance vNIC framework for hypervisor-based NFV with userspace vswitch. Em *Fourth European Workshop on Software Defined Networks*, p. 43–48.

Oracle (2017). Virtualbox. https://www.virtualbox.org - Acessado em dezembro de 2017.

Rasmusson, L. e Corcoran, D. (2014). Performance overhead of KVM on linux 3.9 on ARM cortex-a15. *ACM SIGBED Review*, 11(2):32–38.

Squid (2017). Squid 3 proxy. http://www.squid-cache.org/Intro - Acessado em dezembro de 2017.

Stress (2014). Stress for linux. https://people.seas.harvard.edu/ apw/stress - Acessado em dezembro de 2017.

Technology, A. S. . (2013). Caching improves user experience in 4G LTE networks. http://www.antennasonline.com/main/blogs/caching-improves-user-experience-in-4g-lte-networks - Acessado em dezembro de 2017.

Tsetse, A. K., Wijesinha, A. L., Karne, R. K. e Loukili, A. (2012). A 6to4 gateway with co-located NAT. Em *IEEE International Conference on Electro/Information Technology (EIT)*, p. 1–6.

VirtIO (2017). Virtio - paravirtualized drivers for KVM/linux. http://www.linux-kvm.org/page/Virtio - Acessado em dezembro de 2017.

VMware (2012). Vmware - official site. https://www.vmware.com/ - Acessado em dezembro de 2017.

Yang, J. e Lan, Y. (2015). A performance evaluation model for virtual servers in KVM-based virtualized system. Em *IEEE International Conference on Smart City/SocialCom/SustainCom*, p. 66–71.