Universidade do Estado do Rio de Janeiro
Centro de Tecnologia e Ciências
Faculdade de Engenharia

Rodrigo Silva Vilela Eiras

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

Rio de Janeiro
2017

Rodrigo Silva Vilela Eiras

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

Dissertação apresentada, como requisito parcial para obtenção do título de Mestre em Engenharia Eletrônica, ao Programa de Pós-Graduação em Engenharia Eletrônica, da Universidade do Estado do Rio de Janeiro. Área de concentração: Redes de Telecomunicações.

Orientador: Prof. D.Sc. Marcelo Gonçalves Rubinstein
Orientador: Prof. D.Sc. Rodrigo de Souza Couto

Rio de Janeiro
2017

CATALOGAÇÃO NA FONTE

UERJ / REDE SIRIUS / BIBLIOTECA CTC/B

E34 Eiras, Rodrigo Silva Vilela.
Avaliação de desempenho de um Proxy HTTP implementado como função virtual de rede / Rodrigo Silva Vilela Eiras. – 2017.
157f.

Orientadores: Marcelo Gonçalves Rubinstein e Rodrigo de Souza Couto.
Dissertação (Mestrado) – Universidade do Estado do Rio de Janeiro, Faculdade de Engenharia.

1. Engenharia Eletrônica. 2. HTTP (Protocolo de rede de computação) - Teses. 3. World Wide Web (Sistema de recuperação da informação) - Teses. 4. Virtualização – Teses. I. Rubinstein, Marcelo Gonçalves. II. Couto, Rodrigo de Souza. III. Universidade do Estado do Rio de Janeiro. IV. Título.

CDU 004.5

Assinatura

Data

Rodrigo Silva Vilela Eiras

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

Dissertação apresentada, como requisito parcial para obtenção do título de Mestre em Engenharia Eletrônica, ao Programa de Pós-Graduação em Engenharia Eletrônica, da Universidade do Estado do Rio de Janeiro. Área de concentração: Redes de Telecomunicações.

Aprovada em 12 de Julho de 2017.
Banca Examinadora:

Prof. D.Sc. Marcelo Gonçalves Rubinstein (Orientador)
Faculdade de Engenharia - UERJ

Prof. D.Sc. Rodrigo de Souza Couto (Orientador)
Faculdade de Engenharia - UERJ

Prof. D.Sc. Alexandre Sztajnberg
Faculdade de Engenharia - UERJ

Prof. D.Sc. Sidney Cunha de Lucena
Departamento de Informática Aplicada - UNIRIO

Rio de Janeiro
2017

DEDICATÓRIA

Dedico esta dissertação à minha esposa Anna Carolina, que é a razão de todos os esforços e lutas de minha vida. Sigo confiante de que todas as escolhas são melhores quando escolhemos juntos.

AGRADECIMENTOS

Agradeço à minha esposa Anna Carolina por todo o carinho, respeito e incentivo a dedicação a este trabalho mesmo quando o momento necessitava de subtração do convívio familiar em detrimento da conclusão dessa etapa. É realmente um privilégio poder dividir um lar e meus anseios com você. Agradeço aos meus pais Maria da Conceição e Eudalicio por todo o incentivo ao longo do meu crescimento pessoal e profissional, em todos os momentos, principalmente nos mais difíceis e desafiadores. Agradeço às minhas irmãs Naiara e Naiana pelas palavras de conforto e incentivo nas diversas conversas entre irmãos que tivemos. Agradecimento especial à tia Leninha por todo o suporte e apoio que nos oferece desde sempre. Muitas conquistas da minha vida não seriam possíveis sem sua ajuda. Agradeço aos amigos da IesBrazil Consultoria, especialmente ao Carlos Eiffel, por todo o apoio e liberação para poder me dedicar a esse trabalho, o qual não seria possível sem sua concordância. Aos amigos Márcio e Mariana pela oportunidade de trabalhar no Rio de Janeiro e poder estreitar nosso convívio familiar. Agradeço também ao Programa de Pós-Graduação em Engenharia Eletrônica da Universidade do Estado do Rio de Janeiro pela oportunidade de poder participar juntamente com outros colegas de momentos únicos de atividades de pesquisa e expansão de conhecimento. Agradeço a todos os professores do PEL que de certa forma doam parte do seu tempo em função do desenvolvimento de novos profissionais de pesquisa e docência. Agradecimento especial aos professores e orientadores Marcelo Rubinstein e Rodrigo Couto, pessoas que se tornaram extremamente importantes e referenciais para minha vida pessoal e profissional, pela forma como conduziram esse trabalho fornecendo orientação com maestria e excelência. Não existem palavras que possam descrever minha gratidão. Aos amigos de classe que juntos superamos as diversas dificuldades oriundas de aulas e trabalhos. Agradeço ainda aos membros da banca examinadora pela disponibilidade e aceite para avaliar esse trabalho. Por fim, um agradecimento a todos que de certa forma participaram, mesmo que indiretamente, desse momento único em minha vida. Muito obrigado!

Don't Stop Believin'
Journey

RESUMO

EIRAS, R. S. V. *Avaliação de Desempenho de um Proxy HTTP Implementado como Função Virtual de Rede*. 2017. 66 f. Dissertação (Mestrado em Engenharia Eletrônica) – Faculdade de Engenharia, Universidade do Estado do Rio de Janeiro, Rio de Janeiro, 2017.

Virtualização de Funções de Redes é um novo paradigma em que serviços de redes são virtualizados sobre *hardware* genérico, dispensando o uso de equipamentos específicos para cada serviço de telecomunicações. Nesse tipo de abordagem, um dos principais desafios é o desempenho quando comparado com as soluções proprietárias, e muitas vezes consagradas, que estão disponíveis no mercado. Assim, é importante escolher corretamente a tecnologia de virtualização a ser empregada em uma determinada função de rede buscando a obtenção de um desempenho aceitável. Além disso, a escolha da tecnologia permite um melhor aproveitamento dos benefícios providos pela virtualização como, por exemplo, a flexibilidade e a escalabilidade. Para auxiliar nessa escolha, esta dissertação apresenta uma avaliação de desempenho de duas soluções de virtualização que podem ser aplicadas em NFV, o KVM e o Docker, quando utilizadas para implementar um *proxy* HTTP virtualizado. O KVM é uma plataforma de virtualização tradicional, empregando conceitos de virtualização completa e de para-virtualização, enquanto o Docker implementa uma virtualização leve através de contêineres. Os resultados mostram que o Docker possui desempenho superior ao do KVM, independentemente do tipo de virtualização implementado nesse último. Assim, o Docker apresenta tempos de processamento de requisições HTTP próximos ao de uma solução sem virtualização, o que é um requisito inicial considerado pela indústria de telecomunicações ao se utilizar de ambientes virtualizados para implementar funções de rede. Entretanto, em situações nas quais a flexibilidade e o isolamento são importantes, o KVM pode ser mais adequado, uma vez utiliza uma camada de hipervisor para prover o isolamento completo da instância virtual. Nessa linha, este trabalho mostra também que o uso de para-virtualização no KVM melhora significativamente o desempenho, mas não o suficiente para superar o Docker nos tempos de processamento. Assim, caso seja exigido um maior isolamento e seja tolerável certa queda de desempenho, a para-virtualização do KVM é uma alternativa ao Docker e sua virtualização por contêineres.

Palavras-chave: NFV; *Proxy*; KVM; Docker;

ABSTRACT

EIRAS, R. S. V. *Performance Evaluation of an HTTP Proxy Implemented as Virtual Network Function.* 2017. 66 f. Dissertação (Mestrado em Engenharia Eletrônica) – Faculdade de Engenharia, Universidade do Estado do Rio de Janeiro, Rio de Janeiro, 2017.

As Network Functions Virtualization (NFV) does not require specific *hardware* for each telecommunication service, the main challenge of this approach is the performance when compared to proprietary solutions. Thus, it is important to correctly choose the virtualization technology to be employed in a given network function in order to achieve good performance. Moreover, the right choice of technology allows better utilization of the benefits provided by virtualization, such as flexibility and scalability. To assist in this choice, this work presents a performance evaluation of two virtualization solutions that can be applied in NFV, KVM and Docker, when used to implement a virtualized HTTP proxy. KVM is a traditional virtualization platform, employing complete virtualization and para-virtualization concepts, while Docker implements lightweight virtualization across containers. The results show that Docker performs better than KVM, regardless of the type of virtualization implemented. Thus, Docker can achieve processing times for HTTP requests close to that of a non-virtualized solution, which is an initial requirement considered by the telecommunications industry when using virtualized environments to deploy network functions. However, in situations where flexibility and isolation are important, KVM may be more suitable since it uses a hypervisor layer to provide complete isolation of the virtual instance. This paper also shows that the use of para-virtualization in KVM improves performance, but not enough to overcome Docker. Thus, if better isolation is required and some performance loss is tolerable, KVM with para-virtualization is an alternative to Docker and its container virtualization.

Keywords: NFV; Proxy; KVM; Docker;

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

ACL	*Access Control List*
API	*Application Program Interface*
AUFS	*Advanced multi-layered Unification Filesystem*
CAN	*Campus Area Network*
CPU	*Central Processing Unit*
E/S	*Entrada/Saída*
ETSI	*European Telecommunication Standard Institute*
FTP	*File Transfer Protocol*
HTTP	*HyperText Transfer Protocol*
IO	*Input and Output*
IoT	*Internet of Things*
IDC	*International Data Corporation*
IP	*Internet Protocol*
KVM	*Kernel-based Virtual Machine*
LXC	*LinuX Containers*
NAT	*Network Address Translation*
NFV	*Network Functions Virtualization*
NSF	*National Science Foundation*
OS	*Operating System*
PaaS	*Platform-as-a-Service*
PID	*Process IDentifier*
QoS	*Quality of Service*
RAM	*Random Access Memory*
SDN	*Software Defined Network*
SO	*Sistema Operacional*
TRELIS	*The REduction with LIne Solution*
VNF	*Virtualized Network Function*
WiFi	*Wireless Fidelity*

SUMÁRIO

INTRODUÇÃO

Tradicionalmente, funções de rede como *firewalls*, roteadores e *proxies* são atreladas a dispositivos dedicados. Isso torna o gerenciamento de rede menos flexível e faz com que a topologia de rede tenha uma capacidade de expansão limitada, principalmente pela limitação de espaço físico necessário para abrigar cada vez mais equipamentos necessários ao processamento de altos volumes de dados. 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 uma infraestrutura virtualizada, executando em servidores de uso comum e genérico. A NFV vem atraindo a atenção da indústria de telecomunicações, em função da possibilidade de redução de custos de operação e de manutenção. Assim, evita-se o modelo conservador utilizado em centros de dados em que determinada função de rede é dedicada a um *appliance* de *hardware* específico, proprietário e de custo elevado. Nessa linha, o ETSI (*European Telecommunication Standard Institute*) (ETSI, 2013b) vem publicando uma série de especificações que padronizam uma arquitetura NFV.

Uma questão levantada pela indústria de telecomunicações corresponde ao desempenho de uma função de rede virtualizada (*Virtualized Network Function* - VNF) 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 convidado instalado em uma máquina virtual necessitam ser traduzidas na camada de virtualização, também chamada de 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 sistema operacional, bem como sua configuração de *hardware* virtual dedicado. Isso pode acarretar, em certas condições, um desperdício de recursos quando a máquina virtual estiver ociosa. Utilizando a paravirtualização, um ganho de desempenho é notado, uma vez que o sistema operacional convidado da máquina virtual consegue, através de modificações, acessar diretamente o *hardware* físico do computador. No entanto, questões como quantidade de memória e 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 baseados em *software* executando Linux conseguem atingir uma taxa de encaminhamento de pacotes em torno de 1,2 M pacotes/s. Já em um roteador Linux virtualizado com o Xen (BARHAM et al., 2003), essa taxa atinge aproximadamente 0,2 M pacotes/s. Dessa forma, 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 a nível de processo, ao invés de todo o Sistema Operacional (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, mas com um isolamento fraco entre as VNFs criadas no mesmo ambiente (CZIVA et al., 2015). Assim, devido às questões de isolamento, contêineres podem não ser adequados para aplicações que demandem uma segurança maior, como maior isolamento entre as VNFs.

Uma das funções que podem ser implementadas como VNF é o *proxy*. Um *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. Entretanto, de acordo com (SQUID, 2017), serviços de *proxy* são bastante sensíveis ao nível de utilização do processador (CPU) e isso pode se tornar um empecilho quando esse tipo de função de rede necessita ser virtualizada.

Motivação e Objetivos

Uma vez que sistemas de *proxies* HTTP (HyperText Transfer Protocol) são funções de rede de importância relevante nas atuais topologias de rede, o desempenho esperado por essas aplicações é sempre observado com muita atenção por administradores de sistemas, já que o serviço é um dos corresponsáveis pela qualidade de navegação na *web* e faz parte de modernas topologias de redes de provedores de serviços de acesso à Internet.

Em tempos de tecnologia de computação em nuvem, termos como elasticidade, escalabilidade e flexibilidade na implantação de um recurso virtual estão em voga. Na avaliação de uma suposta contratação ou criação de uma aplicação em ambiente virtual, é avaliado como o desempenho dessas aplicações é afetado adicionando duas ou mais instâncias para distribuir carga de serviços e melhorar os tempos de processamento da aplicação em questão. Por isso, é importante avaliar como as tecnologias de virtualização

podem trabalhar sob esses preceitos.

Outro ponto importante quando se compara NFV, que tem como proposta a utilização de *hardwares* genéricos, com sistemas construídos sobre *appliances* de *hardware* específicos é com relação à atualização da tecnologia. Em ambientes virtualizados, é possível que a atualização seja planejada e programada, tanto em questões de *hardware* quanto em questões de *software* com um custo consideravelmente menor em comparação com soluções proprietárias (COMBE; MARTIN; PIETRO, 2016).

Dado as questões expostas, a proposta dessa dissertação é avaliar de que forma a virtualização tradicional e a virtualização leve, influenciam no desempenho de VNFs de *proxy* HTTP. A avaliação de desempenho é realizada utilizando-se duas plataformas de virtualização caracterizadas como *softwares* livres. O KVM (*Kernel-based Virtual Machine*) (KVM, 2017) é, por padrão, uma plataforma de virtualização completa, enquanto que o Docker (DOCKER, 2017a) implementa virtualização leve através de contêineres. Para realizar os experimentos propostos, serviços do *proxy* Squid 3 (SQUID, 2017) são instanciados como VNFs no KVM e no Docker. Como base de comparação, utilizam-se *proxies* executando diretamente no Linux nativo, isto é, sem utilizar virtualização. Os experimentos mostram que VNFs executando no Docker conseguem ter um desempenho próximo ao do Linux nativo quando operando em um ambiente com carga elevada de sistema.

Nos resultados obtidos, é possível observar que VNFs construídas sobre contêineres conseguem ter um desempenho próximo ao de um sistema Linux nativo. 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 considerada, com um desempenho aproximadamente 50% superior em relação ao KVM com virtualização completa. Adiante nos experimentos, é possível perceber que, tanto para o Docker quando para o KVM, o uso de duas instâncias de *proxy* para balancear o tráfego HTTP das requisições pode melhorar consideravelmente os tempos de processamento das requisições. Com base nessa conclusão, avalia-se ainda o quanto os tempos de processamento do Docker podem ser melhorados à medida que novas VNFs são adicionadas ao teste. Além do exposto, este trabalho fornece uma discussão sobre os cenários e sobre as tecnologias de virtualização utilizadas. Para tal, foi criado um ambiente utilizando três servidores idênticos nos quais foram configuradas máquinas virtuais sobre o hipervisor KVM (Kernel-based Virtual Machine) (KVM, 2017) e contêineres sobre o Docker (DOCKER, 2017a). Em um primeiro cenário, é feita uma avaliação sobre o comportamento das soluções de virtualização com relação à sua convergência, ou seja, em quanto tempo um serviço pode estar disponível e respondendo requisições de conexão. Na prática, isso é importante pois em um ambiente elástico ou com alta demanda momentânea, o tempo que uma instância virtual leva para entrar *online* é relevante. Posteriormente, é feita uma segunda avaliação em que instâncias de *proxy* são avaliadas respondendo requisições HTTP de determinados

clientes e os tempos de processamento são então comparados. Em todos os casos, foi avaliado o desempenho das soluções de virtualização com o subsistema de *cache* do Squid 3 (SQUID, 2017) habilitado ou não, e posteriormente foi selecionado o melhor cenário para uma avaliação de distribuição de carga entre duas ou mais instância virtuais e avaliar se os tempos podem ser melhorados à medida que novas instância são adicionadas no ambiente.

Organização do Texto

Esta dissertação está organizada da seguinte forma. O Capítulo 1 apresenta os conceitos essenciais relacionados a NFV e as técnicas de virtualização. O Capítulo 2 descreve as soluções de virtualização utilizadas neste trabalho avaliadas para NFV. No Capítulo 3, são apresentados conceitos básicos de um sistema de *proxy* HTTP e características acerca de seu funcionamento. Na sequência, o Capítulo 4 fornece trabalhos que se relacionam com essa dissertação e mostra como ela é posicionada na literatura. No Capítulo 5, são apresentadas a avaliação de desempenho realizada e a discussão de seus resultados. Por último, as conclusões e as possibilidades de trabalhos futuros são apresentadas no Capítulo 6.

1 CONCEITOS FUNDAMENTAIS SOBRE A NFV E AS TÉCNICAS DE VIRTUALIZAÇÃO

Com a finalidade de elucidar o entendimento acerca desta dissertação, neste capítulo serão apresentados os conceitos essenciais a respeito da NFV e as técnicas de virtualização que podem ser utilizadas na implementação. Como a técnica de virtualização que for utilizada para implementar uma infraestrutura de NFV pode impactar no desempenho das funções de rede virtualizadas, é importante distinguir e discutir as características de cada uma delas.

1.1 Arquitetura da Virtualização de Funções de Rede

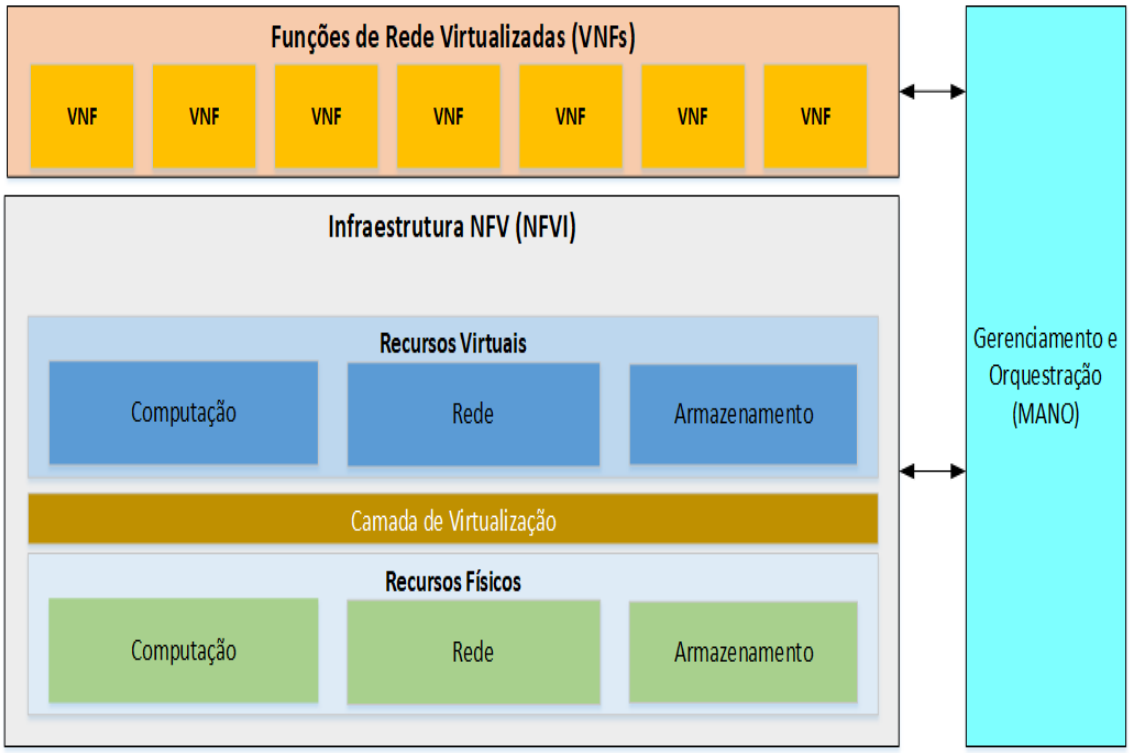
Por definição, a NFV busca remover a relação de dependência entre o *software* e o *hardware* através de alguma técnica de virtualização, permitindo que as funções de redes (NFs) sejam implementadas em computadores de arquitetura genérica e de menor custo (HAN et al., 2015). Para tal, é empregada a virtualização de *software* para criar as funções de rede virtualizadas (VNFs). A infraestrutura de rede física também é virtualizada construindo enlaces virtuais sobre os enlaces físicos e máquinas virtuais (VMs) sobre os servidores (MIJUMBI et al., 2015). O enlace virtual é uma conexão lógica ponto a ponto, embora possa haver vários roteadores entre eles na mesma topologia física. As VMs são abstrações de *software* de uma máquina física, nas quais um sistema operacional completo e genérico pode ser instalado. Isso permite que diversas VMs compartilhem do mesmo *hardware*.

A Figura 1 mostra a arquitetura NFV conforme as definições do Instituto Europeu de Padrões de Telecomunicações (*European Telecommunications Standards Institute*, ETSI), que geralmente é utilizada pelos fabricantes de soluções para NFV e serve como base para os esforços de padronização na área. Na sequência, detalham-se os componentes dessa arquitetura (ETSI, 2013a).

1.1.1 Infraestrutura NFV

A Infraestrutura NFV (*NFV Insfrastructure*, NFVI) é o componente base da arquitetura do *framework* da NFV e é composta por um conjunto de recursos computacionais, de rede e de armazenamento físicos e virtuais. A NFVI consiste em uma coleção de recursos físicos e virtuais, estando relacionada aos dispositivos físicos e VMs que recebem as VNFs, além de envolver também os enlaces físicos e virtuais. A conexão de rede entre os

Figura 1 - Arquitetura da NFV.



Fonte: Adaptado de (ETSI, 2013a).

elementos e os dispositivos de rede também são considerados como parte da NFVI (QUEIROZ; COUTO; SZTAJNBERG, 2017).

Em NFV, a camada de virtualização cria os recursos virtuais, que são abstrações dos recursos de *hardware*, e desacopla as VNFs dos equipamentos físicos. Os recursos físicos são segmentados, isolados e cada parte é destinada a uma VM. A camada de virtualização permite ainda que as VNFs consigam utilizar a infraestrutura virtual subjacente, além de prover os recursos necessários à execução das funções (ETSI, 2013a).

1.1.2 Funções de Rede Virtualizadas

A NFV oferece uma alternativa ao uso de equipamentos dedicados e específicos (*appliances*), através da alocação de VNFs (*Virtualized Network Functions*, VNFs). Essas funções virtualizadas são módulos de *software* que oferecem os serviços dos *appliances* em servidores com *hardware* genéricos que é comumente encontrado em qualquer centro de dados de uma companhia. Dentre os serviços ofertados, estão incluídos NAT, *proxies*, balanceamento de carga, *firewalls*, entre outros. As VNFs são implementações em *software* das funções de rede muitas vezes comercializadas sobre *appliances* que são então habilitadas para funcionar sobre a NFVI, conforme mostra a Figura 1.

Embora o uso de VNFs seja um dos principais pontos a favor da NFV, a virtualização das NFs não é uma imposição. Como o ponto principal é o desacoplamento entre *hardware* e *software*, é possível usar NFs dos fabricantes comerciais instaladas diretamente na máquina física, sem usar máquinas virtuais. O desacoplamento não necessariamente requer a virtualização. Sendo assim, as funções de rede continuam independentes do *hardware*, já que podem ser instaladas em servidores genéricos, mas os benefícios oriundos da virtualização, como a elasticidade e a dinamicidade na alocação de recursos, não seriam contemplados. Além disso, também é possível construir cenários híbridos, nos quais há VNFs executando em recursos virtuais e NFs executando em recursos físicos (MOENS; TURCK, 2014a).

1.1.3 Gerenciamento e Orquestração

Gerenciamento e Orquestração (*Management ANd Orchestration*, MANO) tem relação com os recursos da arquitetura da NFV, responsável pela gestão dos recursos físicos e de *software* que suportam a virtualização da infraestrutura, bem como por gerenciar o ciclo de vida das VNFs. Por isso, esse módulo representado na Figura 1 interage tanto com a NFVI quanto com as VNFs. O MANO se destina a todas as tarefas de gerenciamento da virtualização necessárias na arquitetura NFV (ETSI, 2013a).

A função base do MANO é o de Gerenciador de Infraestrutura Virtual (*Virtual Infrastructure Manager*, VIM), que envolve as funções de controle e gerenciamento da interação entre VNF e os recursos de computação, rede e armazenamento da NFVI. O VIM também é responsável pela virtualização dos recursos físicos, como um hipervisor que provê as abstrações de VMs a sistemas operacionais para que esses utilizem os recursos de *hardware* de uma máquina. É possível ainda um cenário com múltiplos VIMs, o que culmina em várias infraestruturas de NFV, ou seja, podem existir diferentes provedores de infraestrutura, cada um com um VIM designado. O gerenciamento de recursos também é feito pelo VIM, que faz um inventário das capacidades e recursos da NFVI e decide quanto à alocação de máquinas virtuais. O VIM também coleta informações sobre problemas na infraestrutura, planejamento da capacidade, monitoramento e otimização (ETSI, 2013a). Sendo assim, é o VIM que detecta a ocorrência das falhas e deve alertar aos outros blocos sobre o problema, se necessário. Existem outras funcionalidades que são exercidas pelo MANO, como por exemplo o gerente de VNFs, responsável pelo ciclo de vida das VNFs, o repositório de dados onde ficam armazenadas as listas e catálogos contendendo serviços, recursos e capacidades da VNFs e por último o orquestrador responsável por definir o acesso e o compartilhamento de recursos entre as VNFs (ETSI, 2013a).

Apesar do *framework* para implementação de NFV contemplar diferentes módulos e funcionalidades, esta dissertação se posiciona somente na camada de virtualização, a

fim de avaliar o impacto de diferentes técnicas de virtualização quando implementando um *proxy* HTTP como função de rede virtualizada.

1.2 Técnicas de Virtualização de Redes

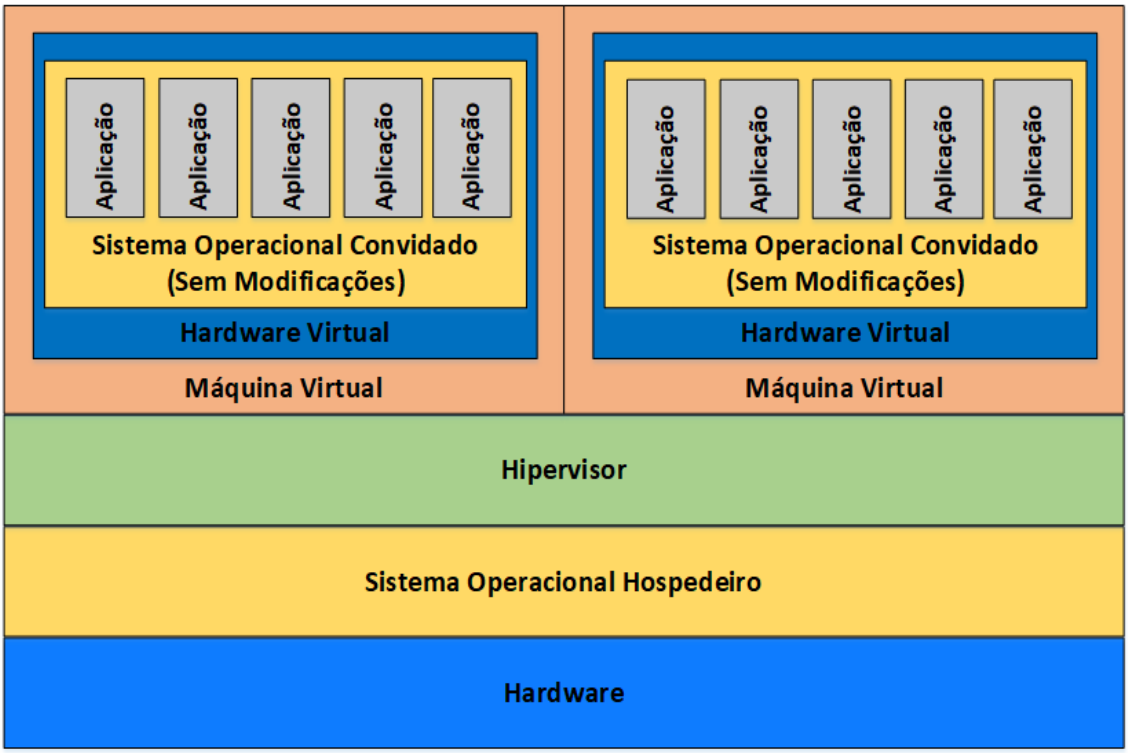
Soluções de virtualização multiplexam o acesso ao *hardware*, permitindo a criação de múltiplas instâncias virtuais sobre uma mesma máquina física. Como a NFV também se baseia na multiplexação do *hardware* de rede, VNFs podem ser executadas sobre soluções de virtualização originalmente concebidas para centros de dados em ambientes corporativos. Neste trabalho, essas soluções são classificadas em três categorias: virtualização completa, para-virtualização e virtualização leve ou conteinerização, e são discutidas com a finalidade de fomentar conhecimento necessário ao entendimento da avaliação de desempenho realizada.

1.2.1 Virtualização Tradicional

Soluções de virtualização tradicionais utilizam hipervisores para multiplexar o acesso ao *hardware*. Para tal, implementam uma camada de *software* chamada 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 sistema operacional hospedeiro a abstração das máquinas virtuais, provendo assim o isolamento entre essas. Dessa forma, máquinas virtuais podem utilizar diferentes sistemas operacionais, chamados de convidados, e consequentemente podem ser configuradas para executar diferentes aplicações. Pode-se afirmar que cada máquina virtual possui sua própria abstração de *hardware* virtual (YILE, 2016) e seu próprio sistema operacional, 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 tradicional, a implementação da virtualização pode ser total, chamada de virtualização completa, ou parcial, chamada de para-virtualização. Esses dois tipos diferem-se em relação às chamadas que passam pelo hipervisor, como detalhado adiante. É importante ressaltar ainda que, independentemente da solução de virtualização a ser utilizada, seja tradicional ou leve, o ambiente de implantação deve ser cuidadosamente estudado e corretamente dimensionado, levando em consideração os serviços e as cargas de utilização que serão impostas às VNFs. É de conhecimento que soluções baseadas em virtualização possuem consideráveis ganhos na economia de recursos e aproveitamento efetivo do *hardware*. No entanto, aumentam a complexidade do gerenciamento da infraestrutura de redes (ETSI, 2013b).

Figura 2 - Arquitetura da virtualização completa.



1.2.1.1 Virtualização Completa

Na virtualização completa, todas as instruções das máquinas virtuais são interceptadas pelo hipervisor antes de chegarem ao *hardware*. Nesse modelo, as aplicações realizam chamadas ao sistema operacional convidado que, por sua vez, faz chamadas a uma abstração virtual de *hardware*, conforme mostrado na Figura 2. Assim, não há necessidade de se alterar qualquer parte do sistema operacional executado em uma máquina virtual (BARHAM et al., 2003). Existem diversas soluções de virtualização completa disponíveis atualmente, tais como VMware ESXi (VMWARE, 2017), Citrix Xen Server (CITRIX, 2017), Microsoft Hyper-V (MICROSOFT, 2017b), Oracle Virtualbox (VIRTUALBOX, 2017), KVM (KVM, 2017) entre outras.

Entre as principais características que envolvem a virtualização completa, estão a capacidade de virtualizar qualquer sistema operacional, de arquitetura suportada, uma vez que não são necessárias modificações no sistema operacional virtualizado. Sabendo que o *hardware* virtual foi criado para suportar determinada arquitetura de um sistema operacional, a instalação de versões legadas de sistemas operacionais também é possibilitada. No entanto, o desempenho geral de uma solução construída nesse tipo de arquitetura fica prejudicado quando comparado ao desempenho em outras variações de arquitetura de virtualização.

Em relação à NFV, a principal desvantagem do uso da virtualização completa 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 às aplicações executadas em ambientes que não usam virtualização (FERNANDES et al., 2011). É importante salientar ainda que, mesmo utilizando virtualização completa, a portabilidade de instâncias virtuais para diferentes *hardwares* físicos, Intel e AMD, por exemplo, pode não ser possível por questões de arquitetura de construção do processador (PEETZ, 2013). A portabilidade ou capacidade de migração de instâncias virtuais em diferentes arquiteturas de *hardware* é relevante, pois em um ambiente de centro de dados, funções de rede que estejam trabalhando em conjunto (em *cluster)*, por exemplo, podem precisar ser movimentadas, criadas ou removidas de acordo com a demanda de utilização. A preocupação com a interoperabilidade da solução, independente do *hardware* onde a instância virtual será alocada, é de fato importante.

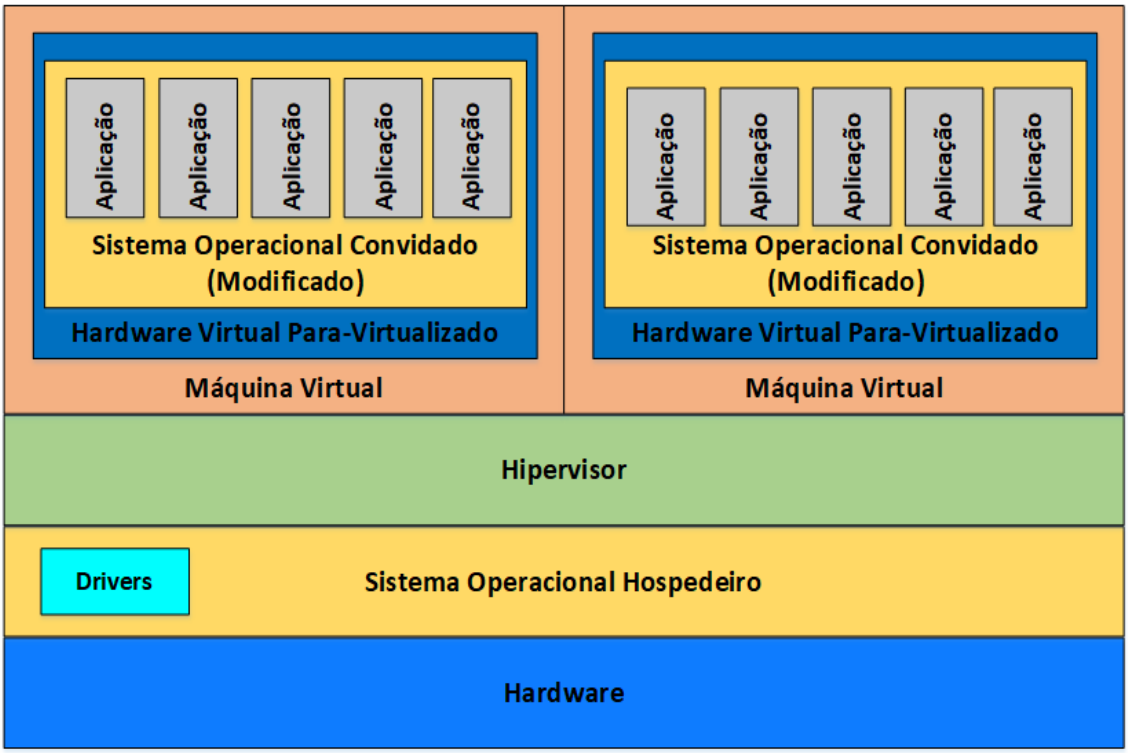
1.2.1.2 Para-Virtualização

O conceito de para-virtualização, também denominada virtualização parcial, é similar ao 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 podem ser implementadas sobre esse hipervisor. A diferença está no fato de que o sistema operacional da máquina virtual (SO convidado) está "ciente" de sua virtualização, podendo assim realizar chamadas diretamente ao *hardware*, melhorando o desempenho (CINTRA, 2010). A IBM, a VMware, a Red Hat e a Citrix, reconhecidas por seus virtualizadores eficientes, possuem em seus portfólios de produtos soluções que implementam para-virtualização. Existem ainda soluções gratuitas que podem implementar um ambiente para-virtualizado, como por exemplo o VirtualBox (VIRTUALBOX, 2017) da Oracle e o KVM (KVM, 2017), solução de código aberto que é mantida por uma comunidade de *software* livre.

Conforme ilustrado na Figura 3, na para-virtualização os sistemas operacionais convidados não utilizam uma abstração de *hardware* virtual.

Para que a comunicação seja possível, o sistema operacional convidado requer que modificações ou extensões sejam aplicadas no momento da implementação da virtualização para que esses possam fazer chamadas de sistemas diretamente ao *hardware*. Em outras palavras, diferentemente da virtualização completa, a para-virtualização precisa de alterações no sistema operacional da máquina virtual para, a partir disso, poder se beneficiar com ganhos de desempenho. Na para-virtualização há limitações acerca da integração hospedeiro e convidado, já que é necessário modificar o sistema operacional

Figura 3 - Arquitetura da para-virtualização.



das máquinas virtuais. Em alguns casos, essa modificação possui complexidade proibitiva. As alterações 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.

Existem quatro níveis (também chamados de anéis) de instruções privilegiadas que são executadas pelo sistema operacional em modo protegido, numeradas de 0 a 3, sendo que 0 corresponde às instruções com maior privilégio e 3 às instruções com menor privilégio. Isso quer dizer que operações de E/S (Entrada/Saída) e chamadas de *drivers* de dispositivos são realizadas no nível 0 e aplicações são executadas no nível 3 (CONDURACHE et al., 2015). Um exemplo disso são as instruções privilegiadas de nível 0 que, para acessar certos setores da memória RAM ou realizar chamadas de *drivers* de dispositivos de E/S (Entrada/Saída), são executadas diretamente no *hardware* (BABU et al., 2014). É importante notar que tanto na virtualização completa quanto na para-virtualização, a camada de hipervisor é presente na arquitetura sendo fundamental para o funcionamento das máquinas virtuais. Além disso, independentemente de a para-virtualização considerar a modificação de *drivers*, o isolamento da máquina virtual e do seu sistema operacional é completo.

Grande parte dos ambientes nos quais se encontram soluções para-virtualizadas se baseiam no Linux. Somado ao fato deste 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.

1.2.2 Virtualização Leve ou Conteinerização

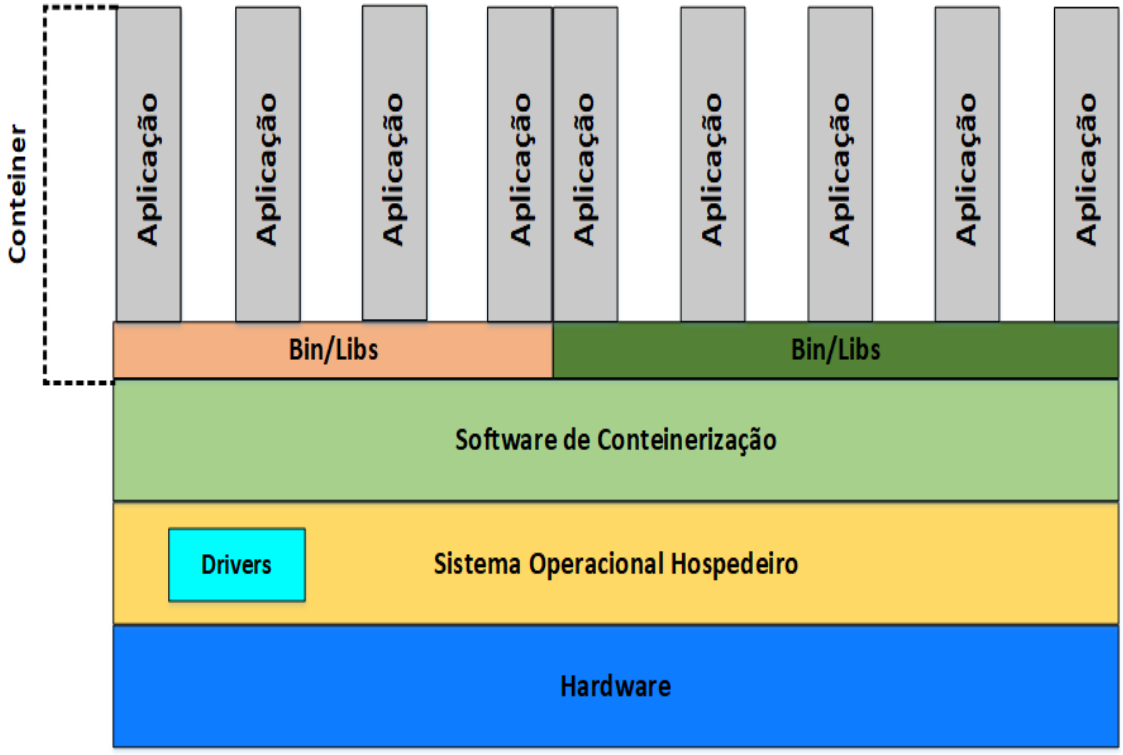
A virtualização leve, também denominada conteinerização, se refere à técnica de virtualizar aplicações utilizando contêineres. Diferentemente da virtualização completa e da para-virtualização, contêineres implementam o isolamento entre as aplicações a nível de processos no sistema operacional base, evitando assim o *overhead* imposto pela abstração de *hardware* do hipervisor nas máquinas virtuais. Conforme ilustrado na Figura 4, contêineres compartilham um mesmo sistema operacional instalado na máquina física e uma ou mais aplicações podem ser executadas dentro de um ou mais contêineres. Para tal, contêineres fornecem instâncias de espaços de usuário (*user spaces*) de forma isolada. Muitos desses sistemas de contêineres são derivados da tecnologia de contêineres LXC (CONTAINERS, 2017), que utilizam *cgroups* (*Linux Control Groups*) e tecnologias de *namespacing* para prover recursos de gerenciamento e de isolamento entre os contêineres.

O *CGroups* ou *Control Groups*, é responsável pelo controle de uso dos recursos por processo ou grupo de processos, podendo assim executar diferentes contêineres com diferentes limites de utilização, por exemplo memória, CPU e E/S. Já o recurso de *Namespaces* possibilita a abstração de processos dentro do *kernel*, isso quer dizer que um processo ou grupo de processo é isolado dentro do *kernel*, podendo visualizar os pontos de montagem, id de processos, id de usuário, *hostname* e a fila de processo, isolados de outros processos ou grupo de processo. Devido a essas características é que chamamos de contêiner todo sistema criado utilizando essa tecnologia, pois o sistema fica "enjaulado" dentro de uma caixa de recursos alocados exclusivamente para ele, e claro limitado a esses recursos (DOCKER, 2017a).

Assim como nos processos do Linux, *cgroups* são organizados hierarquicamente de forma que nós inferiores de *cgroups* podem herdar atributos de nós superiores. A principal diferença é que, diferentemente de processos, *cgroups* não são parte de uma mesma (*thread*) e com isso muitos deles podem existir de forma simultânea no ambiente. Cada *cgroup* pode ser utilizado para gerenciar recursos de um grupo de processos; por exemplo, a utilização de processador e memória pode ser limitada dentro de um *cgroup*. Dessa forma, *cgroups* podem ser utilizados para limitar os recursos que um contêiner pode consumir. Da mesma forma, uma vez que um *cgroup* é removido do sistema, os contêineres associados a ele são finalizados de forma análoga à utilização do comando *kill* do Linux (RAHO et al., 2015).

O Linux provê *namespaces* para *interfaces* de rede, pontos de montagem do sis-

Figura 4 - Arquitetura da virtualização leve.



tema de arquivos, usuários e grupos utilizadores do sistema operacional, *PIDs* (Process IDentifiers) e de *hostname*. Cada um desses *namespaces* oferece recursos que podem ser utilizados pelos contêineres para isolar aplicações que não estão implementadas dentro do mesmo contêiner ou do mesmo *namespace*. Um exemplo prático desse esquema de isolamento corresponde ao usuário criado dentro do contêiner que não tem qualquer ação no ambiente externo ao qual foi criado (BIEDERMAN; NETWORX, 2006).

Dois tipos de contêineres podem ser implementados: contêineres de aplicação e contêineres de sistema operacional. A diferença entre eles está no fato de que o segundo é configurado para executar todo o ambiente do sistema operacional, com variáveis de ambientes próprias e execução dos níveis de inicialização ao iniciar o sistema operacional dentro do contêiner. Já com relação aos contêineres de aplicação, esses oferecem um ambiente restrito e controlado para execução somente da aplicação a ser virtualizada. É recomendado, sempre que possível, a utilização de contêineres orientados a aplicação, nos quais não há desperdício de recursos alocados desnecessariamente. Diferentes soluções no mercado implementam tecnologias de contêineres, sendo as mais destacadas, o OpenVZ (VIRTUOZZO, 2017), o CoreOS (COREOS, 2017) e o Docker (DOCKER, 2017a).

O compartilhamento do *kernel* do sistema operacional hospedeiro entre os contêineres gera vantagens como, por exemplo, capacidade de provisionamento de grande quantidade de instâncias de uma aplicação em um curto espaço de tempo. Isso ocorre, pois, as instâncias não necessitam de imagens dedicadas de disco rígido virtual, ao contrário do

caso da virtualização tradicional. Entretanto, devido ao compartilhamento do *kernel* e dos *drivers* de dispositivos entre os contêineres, não é possível executar diferentes sistemas operacionais em uma mesma máquina física.

Outra desvantagem ao se implantar a tecnologia de contêineres é o método de isolamento provido por essas soluções. Uma vez que o *kernel* do Linux é compartilhado com os contêineres, podem ocorrer problemas de segurança que afetam todo o ambiente no qual a solução de contêiner está implementada (BUI, 2015).

Com relação à aplicação de contêineres em NFV, publicações da literatura (FELTER et al., 2015), (COMBE; MARTIN; PIETRO, 2016), (BONDAN; SANTOS; GRANVILLE, 2016) indicam que o desempenho gerado por soluções utilizando contêineres é um fator motivacional para o desenvolvimento de pesquisa nesse sentido. No entanto, alguns desafios ainda permeiam o assunto, como o desempenho, portabilidade, elasticidade e escalabilidade das soluções de contêineres em NFV.

2 SOLUÇÕES DE VIRTUALIZAÇÃO AVALIADAS

Diferentes soluções de virtualização disponíveis no mercado podem ser utilizadas para implementação de funções virtuais de redes. Neste trabalho, são determinadas como premissas iniciais da escolha da solução a gratuidade e a característica de ser uma plataforma aberta para desenvolvimento. Nesse sentido, duas soluções foram selecionadas: o KVM (*Kernel-based Virtual Machine*) (KVM, 2017) e o Docker (DOCKER, 2017a). O KVM se enquadra nos casos nos quais a virtualização completa e a para-virtualização são implementadas. O fato de o KVM ser integrado nativamente ao Linux também é um fator diferencial na escolha. Em questões de desempenho e quando comparado com o Xen (PROJECT, 2017), que consiste em uma outra plataforma gratuita largamente utilizada, O KVM tem um desempenho superior no trabalho apresentado em (SORIGA; BARBULESCU, 2013) quando avaliando o tráfego de rede. No trabalho (MAURICIO; RUBINSTEIN, 2015) em que é avaliado o desempenho de uma aplicação de *firewall* em um ambiente virtualizado com KVM e Xen, o KVM obteve melhor desempenho nos casos avaliados.

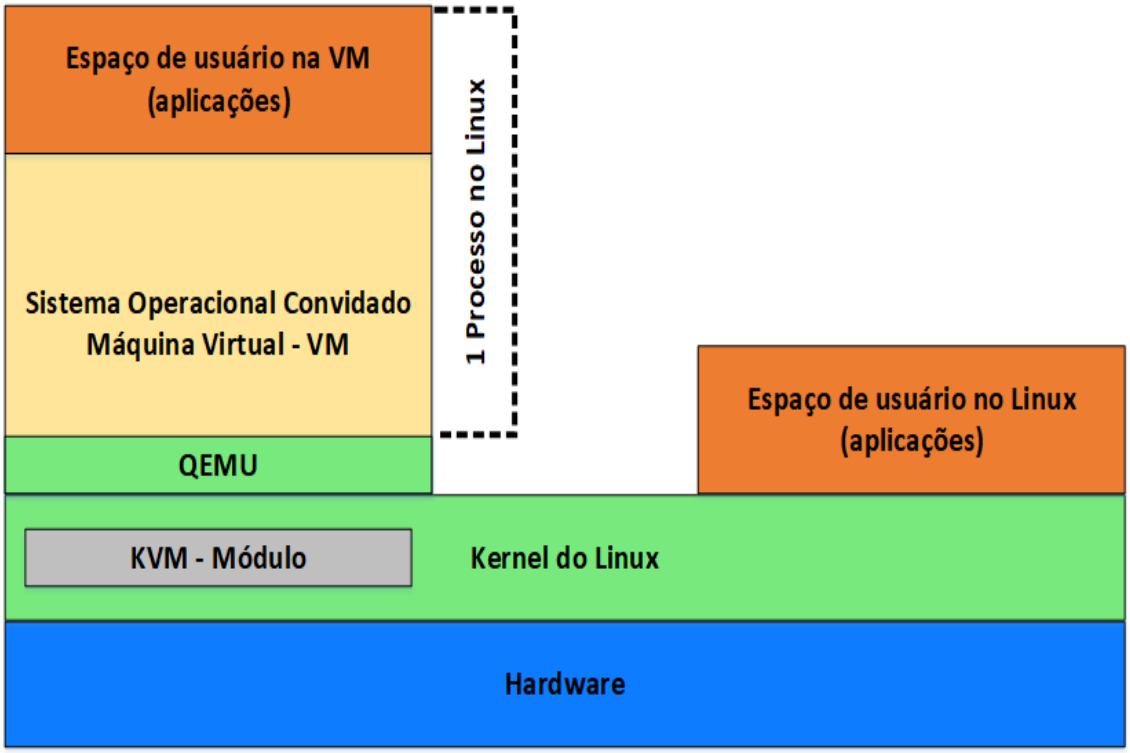
Por fim, segundo (IDC, 2014), o número de servidores Linux está crescendo rapidamente e 72 milhões de novos servidores serão implementados até o final de 2017. Isso é uma grande oportunidade para crescimento do KVM, uma vez que ele já vem junto com a maioria das distribuições Linux, bastando apenas habilitá-lo. O IDC (*International Data Corporation*) afirma ainda que mais de 278.000 novos servidores utilizando KVM foram implementados desde 2011, o que representa uma taxa de crescimento anual de 42% (IDC, 2014).

Com relação aos contêineres, neste trabalho optou-se por utilizar o Docker, que é uma solução de conteinerização que possui grande aceitação e consequente adoção por parte de desenvolvedores para fornecimento de microsserviços. Segundo (DATADOG, 2017) a utilização do Docker pelas empresas cresceu aproximadamente 40% somente no ano de 2016 em relação ao ano anterior. Entre as aplicações mais utilizadas pelas empresas na adoção dos contêineres, a que mais se destaca é o NGINX (NGINX, 2017), popular servidor de páginas *web*.

2.1 A Plataforma KVM

O KVM é uma plataforma de virtualização originalmente desenvolvida pela *Qumranet, Inc* e que em 2007 foi incorporada à linha de desenvolvimento principal do *kernel* do Linux, dando a esse capacidades nativas de virtualização. O KVM, assim como outros *softwares* de virtualização, utiliza extensões de virtualização providas por fabricantes de

Figura 5 - Arquitetura do KVM para ambiente de virtualização completa.

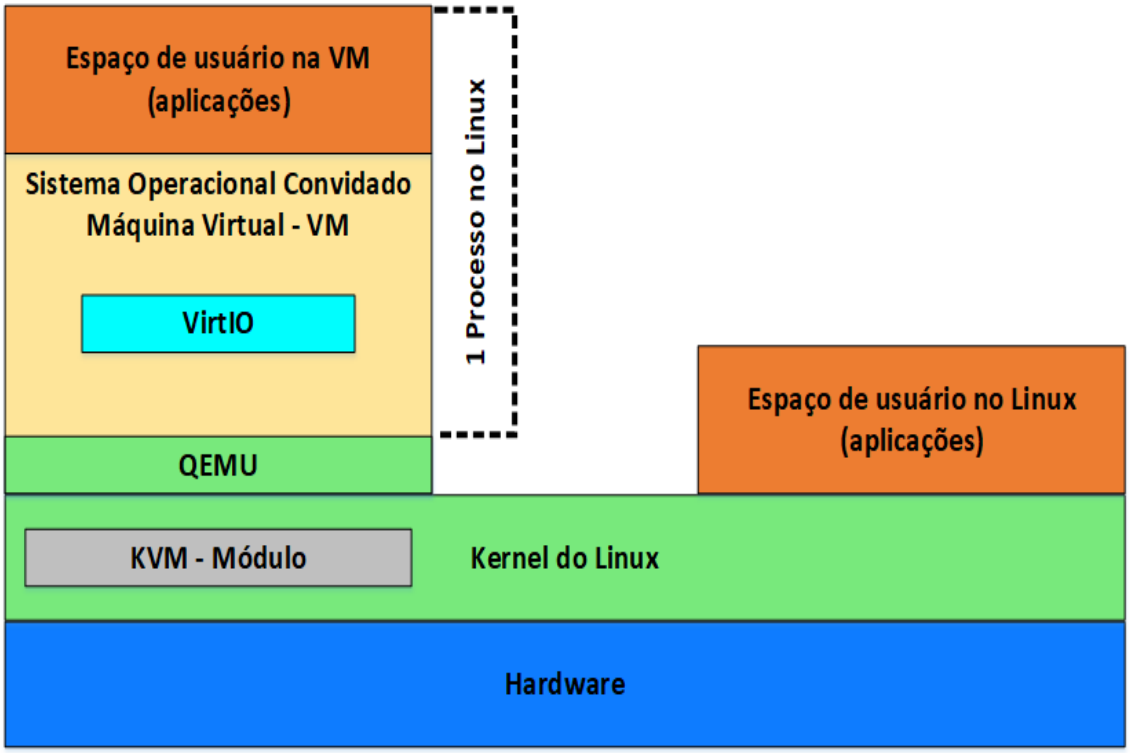


Fonte: Adaptada de (MAURICIO; RUBINSTEIN, 2015)

hardwares, conhecidas como Intel VT-x e AMD-V. Em 2008, a *Qumranet, Inc* foi adquirida pela Red Hat, Inc. que, desde então, oferece uma versão comercial do KVM e suporte ao produto em produção. Uma vez incorporado e acompanhando o desenvolvimento do Linux, o KVM passou a ser um módulo do *kernel* que provê toda a infraestrutura de *software* necessária para conceder ao Linux capacidades de se tornar um hipervisor. A alocação de recursos e a utilização de memória são controladas nativamente pelo *kernel*, enquanto a emulação de dispositivos é controlada por uma versão modificada do QEMU (QEMU, 2017). A máquina virtual é então executada no sistema operacional hospedeiro através do hipervisor, no espaço de usuário (*user space*), e é tratada como um processo comum controlado pelo *kernel* no sistema (MAURICIO; RUBINSTEIN, 2015). Por ser executado na área de usuário, o acesso a funções privilegiadas, por exemplo, no anel 0, deve ser feito através do hipervisor. A Figura 5 mostra a arquitetura da plataforma de virtualização KVM e ilustra o fato de a máquina virtual, juntamente com suas aplicações, ser tratada como um único processo no sistema operacional.

Um processo comum executado no Linux pode ter dois modos de execução: o modo *kernel* e o modo usuário (*user*). Para implementar a virtualização, o KVM adiciona um terceiro modo chamado convidado (*guest mode*). Por operar em modo convidado, todas as máquinas virtuais executadas no KVM são tratadas como processos comuns e independentes dentro do Linux, cada máquina virtual implementada receberá um PID (IBM,

Figura 6 - Arquitetura do KVM com módulo VirtIO para ambiente para-virtualizado.



Fonte: Adaptada de (MAURICIO; RUBINSTEIN, 2015)

2012). Devido ao fato de o Linux tratar cada máquina virtual do KVM como um processo comum no sistema, existe então a possibilidade de se trabalhar com a priorização de um determinado processo em detrimento de outro. Com relação à alocação e ao controle de recursos, o KVM implementa os chamados grupos de controle que determinam o que cada máquina virtual pode ou não consumir. Um exemplo de recurso é a memória RAM (*Random Access Memory*).

O KVM suporta também a para-virtualização utilizando um subsistema chamado VirtIO (VIRTIO, 2017). O VirtIO é um *framework* desenvolvido para acelerar instruções de E/S em um ambiente virtualizado com KVM. De forma simples, o VirtIO fornece uma série de *drivers* e filas compartilhadas entre o sistema operacional convidado e o sistema operacional hospedeiro com a finalidade de diminuir o *overhead* imposto pelo hipervisor (NAKAJIMA; MASUTANI; TAKAHASHI, 2015). A Figura 6 ilustra o posicionamento do módulo VirtIO na arquitetura do KVM. A utilização do VirtIO permite que a máquina virtual alcance um maior desempenho quando comparado com implementações utilizando virtualização completa.

2.2 A Plataforma Docker

O Docker é uma plataforma que provê as funcionalidades de criar e gerenciar contêineres no Linux. Foi criado em 2013 e, em um curto espaço de tempo, acabou se tornando muito popular entre desenvolvedores devido à sua capacidade de implementação simplificada. Originalmente, o Docker foi construído sobre a plataforma de contêineres LXC (CONTAINERS, 2017), provendo uma forma mais fácil de se implementar serviços em contêiner. Diferentemente de contêineres construídos sobre a plataforma LXC, no qual é necessário o fornecimento de uma imagem do Linux para hospedar as aplicações, o Docker permite criar imagens de aplicações que são então armazenadas em seus contêineres. Isso simplificou a forma como contêineres são implementados. A partir da versão 0.9, o Docker inseriu a *libcontainer* (DOCKER, 2017c) em sua plataforma que, assim como o LXC, é uma compilação baseada na arquitetura de *cgroups* e *namespaces* do Linux. O foco da *libcontainer* é a melhoria do isolamento das aplicações instanciadas nos contêineres.

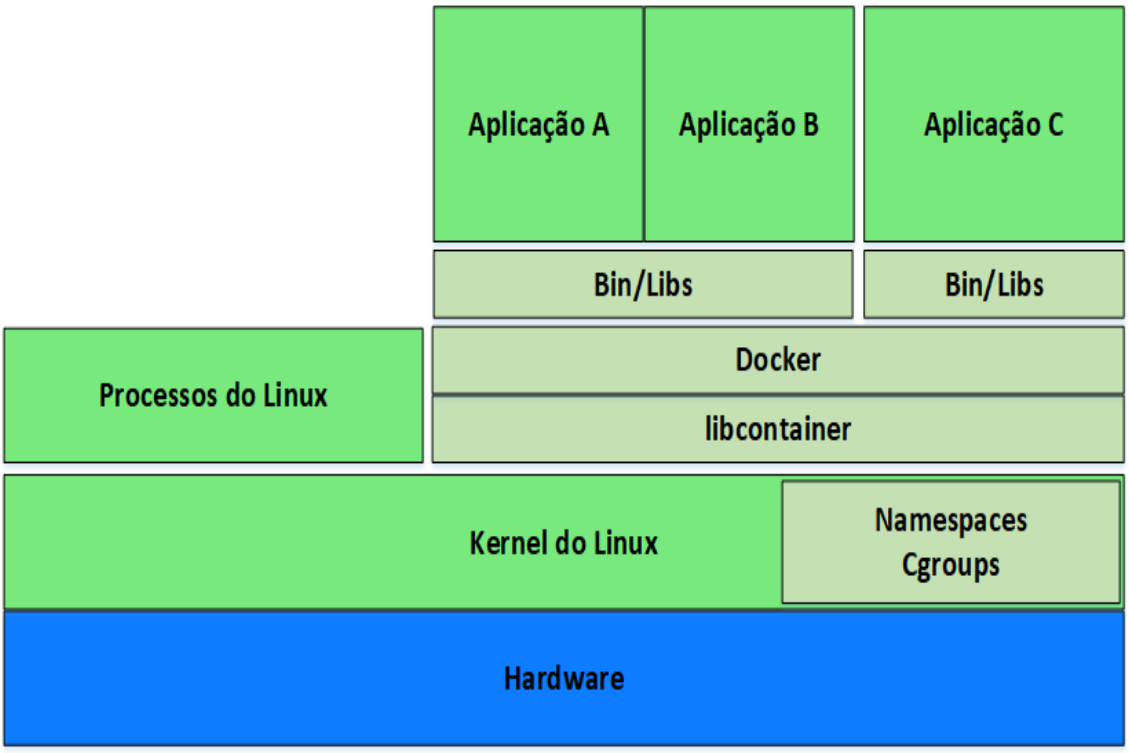
A partir da versão 0.7, o Docker passou a utilizar em sua plataforma um novo tipo de sistema de arquivos para melhorar o desempenho das aplicações dentro de contêineres, chamado de AUFS *(Advanced multi layered Unification FileSystem)*, que funciona basicamente em modo de somente leitura. As aplicações dentro do contêiner utilizam esse sistema de arquivos, protegendo os dados originais de alterações equivocadas ou acidentais. Adicionalmente, uma camada de escrita é adicionada sobre cada contêiner, que permite gravar novas configurações a partir do sistema de arquivo original, através de um comando. Essas gravações podem ser feitas a qualquer momento, mesmo com o contêiner e/ou a aplicação em execução. Caso alterações feitas em memória não tenham sido confirmadas através de um comando para gravação, essas alterações são perdidas na próxima inicialização. Na prática, esse procedimento é importante, pois permite que uma imagem inicial de um contêiner, denominada matriz, seja utilizada por outras implementações sem interferências. Do ponto de vista do gerenciamento, uma vez que a imagem principal da aplicação é atualizada para uma nova versão, todos os demais contêineres que foram criados a partir da matriz poderão ser atualizados diretamente sem a necessidade de atualização unitária. Esse procedimento torna simples o gerenciamento e o rastreamento de problemas em caso de alguma falha. Adicionalmente, o Docker provê um registro público e gratuito de contêineres e aplicações, chamado de Docker Hub (DOCKER, 2017b), no qual é possível baixar imagens de contêineres e aplicações desenvolvidas por outros.

A Figura 7 ilustra a arquitetura do Docker e suas aplicações isoladas através de *cgroups* e de *namespaces*. O Docker e o libcontainer, por sua vez, são considerados processos do Linux e são responsáveis pelo gerenciamento de cada aplicação implementada na plataforma.

Em resumo, os principais atributos do Docker são:

- processo - Cada contêiner recebe um PID e um endereço IP (*Internet Protocol*)

Figura 7 - Arquitetura do Docker.



privado;

- isolamento de recursos - Utiliza os conceitos de *cgroups* e *namespaces*;
- isolamento de rede - Recebe um endereço IP privado em uma *interface* de rede virtual;
- isolamento do sistema de arquivos - Cada contêiner possui o seu próprio sistema de arquivos protegido de gravação acidental e isolado.

Adicionalmente, a Tabela 1 fornece uma comparação acerca das plataformas de conteinerização e das plataformas de virtualização tradicional.

Segundo o portal IW (WEEK, 2016), um grande percentual de empresas começou a experimentar a utilização de soluções baseadas em contêineres em 2016, pesquisa essa que foi realizada pelo site *DevOps.com* através de entrevistas nos meses de abril e maio do mesmo ano e que envolveu 310 empresas participantes. A pesquisa descobriu que 79% das organizações já exploravam de alguma forma a utilização de contêineres e um percentual similar, 76%, estavam utilizando-os em ambientes de produção. Além disso, 52% das empresas entrevistadas disseram que parte do orçamento de tecnologia da informação já era direcionado para soluções que, de alguma forma, simplificam suas operações com contêineres. Das soluções de contêineres mencionadas, o Docker aparece em primeiro lugar com 76% das empresas adotando-o como solução primária de conteinerização. A

Tabela 1 - Comparação entre Conteinerização e Virtualização Tradicional. Adaptado de (DUA; RAJA; KAKADIA, 2014)

Parâmetro	Máquinas Virtuais	Contêineres
Sistema Operacional Convidado	Cada máquina virtual possui seu próprio *hardware* virtual e seu próprio *kernel* é carregado na memória virtual	Todos os contêineres compartilham da mesma memória e do mesmo *kernel* do sistema operacional hospedeiro
Isolamento do Sistema	Área de memória, bibliotecas de sistema e processos são completamente isolados	O *kernel* e seus respectivos módulos são compartilhados entre os contêineres e a máquina física
Desempenho	Todas as instruções necessitam ser traduzidas entre o convidado e o sistema operacional hospedeiro, incorrendo em queda de desempenho	Desempenho aproximado ao da máquina física
Comunicação	*Interfaces* de redes virtuais	*Sockets* e mecanismos nativos do Linux
Armazenamento	Necessitam de bastante espaço em disco para armazenar o sistema operacional virtual e suas aplicações	Necessita de pouco espaço, uma vez que os contêineres compartilham o disco com a máquina física

pesquisa mostra que soluções baseadas em contêineres chegaram para se estabelecer em definitivo, aumentando em muito a expectativa da indústria de telecomunicações pela sua aplicação em NFV.

3 O *PROXY* HTTP COMO FUNÇÃO DE REDE VIRTUALIZADA

A função de rede escolhida para ser avaliada neste trabalho é o *proxy* HTTP. O nome *proxy* é utilizado quando se deseja definir algum serviço intermediário entre o usuário e o servidor no qual ele se conecta. Na prática isso quer dizer que todas as conexões de um determinado serviço que faça uso de um sistema *proxy* terão que passar por esse serviço intermediário. Com a evolução da tecnologia e com a já factível onipresença de usuários em sua relação com a Internet, a utilização de serviços de *proxy* por parte de provedores de serviço de acesso à *web* é cada vez mais justificado. Além de acelerar transferências de objetos requisitados por utilizadores da rede através de um sistema de cache, o sistema de *proxy* também auxilia na economia de banda de navegação, já que em algumas situações não é necessário acessar destinos fora da rede do provedor. A segurança no acesso muitas vezes garante ao provedor a auditoria e o resguardo das ações que por ventura possam ser realizadas de forma maliciosa por seus clientes.

Um sistema de *proxy* HTTP é uma função de rede que visa responder requisições ou solicitações de requisições HTTP no lugar de um servidor *web* e geralmente é instalado em um provedor de serviço para acelerar ou filtrar a conectividade de clientes. No primeiro caso, quando atuando como acelerador de conexões, o sistema de *proxy* é utilizado em conjunto com algum sistema de *cache* para armazenar os objetos requisitados pelos dispositivos conectados a esse. Nesse tipo de configuração, quando um objeto é requisitado por algum cliente esse objeto é então armazenado em uma memória provisória para posteriormente servir de cópia a novas requisições que possam ser enviadas por outros clientes. Com essa operação, o tempo de processamento tende a diminuir, uma vez que a solicitação e a resposta com o objeto a ser transferido podem não precisar passar por enlaces de gargalo existentes entre o cliente e o servidor. No segundo caso, uma solução de *proxy* HTTP também pode muitas vezes operar como um filtro de conexões. Isso significa que, através de ACLs (*Access Control Lists*), diversas configurações a respeito do tipo de conteúdo a ser requisitado pelo cliente podem ser aplicadas. Geralmente os principais filtros criados por administradores de sistemas envolvem:

- o endereço da estação do cliente;
- o domínio requisitado;
- o endereço de rede da origem ou do destino;
- o tipo do objeto requisitado, por exemplo, documentos, músicas, vídeos e outros;
- o período do dia ou da semana, por exemplo, que pode ser concedido o acesso à Internet;

- uma autenticação obrigatória para navegação na Internet, funcionando como uma assinatura digital.

Os filtros mencionados anteriormente podem ser utilizados de forma isolada ou em conjunto, levando em consideração que o processamento de ACLs é sempre sequencial (ISHIKAWA et al., 2011).

A interceptação da requisição pelo serviço de *proxy* HTTP pode ser feita basicamente de duas formas. Na primeira, o cliente deve configurar em seu navegador *web* o endereço do serviço de *proxy* responsável por permitir ou monitorar a conexão. Na segunda forma, chamada de *proxy* transparente, uma configuração pode ser feita no roteador utilizando regras de NAT para interceptar automaticamente o pacote HTTP do cliente, verificá-lo e somente então encaminhá-lo ao destino. Apesar de muito utilizada pelas empresas, a configuração em que o sistema de *proxy* é utilizado de forma transparente é desencorajada, pois quebra o princípio da conexão fim-a-fim entre a origem e o destino. Existem mecanismos que configuram automaticamente o endereço do servidor *proxy* no navegador do cliente sem a necessidade de utilizar regras de NAT para tal (KUROSE; ROSS, 2013). Esses mecanismo são realizados, por exemplo, através de *scripts* e ou políticas de domínio de rede.

A utilização de um serviço de *proxy* em uma rede corporativa pode trazer alguns benefícios, tais como: redução de tráfego de rede e redução de latência, uma vez que o objeto requisitado pode estar armazenado no *cache* do próprio sistema de *proxy* da VNF. Nesse caso, será evitado que a requisição HTTP percorra mais nós em uma topologia de rede até o objeto originalmente requisitado; redução de carga no serviço HTTP de um servidor remoto. Além dos benefícios citados anteriormente, é importante salientar que uma VNF de *proxy* adiciona uma camada extra de segurança, protegendo o cliente de ameaças provenientes da *web* (KUROSE; ROSS, 2013).

Neste trabalho é utilizado o *proxy* Squid 3 (SQUID, 2017), que é uma aplicação conhecida na comunidade Linux, gratuita, de código aberto e é largamente utilizada por diversas empresas e corporações, seja de caráter público ou privado. Em 1990, uma iniciativa coordenada pela NSF (*National Science Foundation*) para criar um sistema de *cache* para conexões HTTP deu início ao desenvolvimento do Squid. Atualmente, e já na versão 3, o Squid suporta nativamente os protocolos de aplicação HTTP e FTP, operando sobre IPv4 e IPv6. Toda a configuração do sistema é realizada utilizando um arquivo padrão denominado *squid.conf*, no qual ACLs são criadas para permitir o tráfego de clientes e também no qual se habilita o subsistema de *cache* disponível como acelerador de requisições.

4 TRABALHOS RELACIONADOS

Diversos trabalhos abordam diferentes tópicos de NFV na literatura. No entanto, nenhum contribui especificamente na implementação de *proxies* HTTP como função de rede virtualizada. Esta seção descreve a literatura relacionada, classificando os estudos em trabalhos genéricos sobre NFV, trabalhos que consideram contêineres em NFV, além de trabalhos genéricos sobre plataforma de virtualização e sistemas de *proxy* HTTP.

4.1 Trabalhos genéricos sobre NFV

Como NFV representa um novo paradigma de implementação de funções de redes, tanto a academia quanto a indústria de telecomunicações se mostram interessadas no desenvolvimento de soluções que possam suportar o avanço da tecnologia nesse sentido. O trabalho (QUEIROZ; COUTO; SZTAJNBERG, 2017) foca o desafio de se posicionar corretamente uma função de rede virtualizada em uma infraestrutura de rede virtual, de forma que ela possa ser resiliente e energeticamente eficiente. Para atingir o objetivo proposto, é formulado um problema de programação inteira mista para que se possa posicionar funções virtuais de rede escolhendo os servidores que atenderão às demandas de serviço. O problema formulado minimiza o uso de energia e provê resiliência às funções de rede virtuais através do compartilhamento e da replicação dessas funções. Para solucionar o problema, é proposta a heurística TRELIS, que reduz o tamanho do problema economizando até 15% de energia quando o compartilhamento de funções virtuais de rede é usado.

No trabalho (KOURTIS; GARDIKIS, 2016) é discutido como as funções virtuais de redes são alocadas muitas vezes em ambientes de larga escala que envolvem diferentes parâmetros e/ou sistemas heterogêneos que podem impactar no desempenho do serviço virtualizado. Muitas vezes uma topologia de rede envolve dezenas ou centenas de funções de rede que podem falhar a qualquer momento. Assim, um mecanismo de monitoramento eficiente que trabalhe proativamente prevendo falhas e corrigindo erros é um desafio e é essencial para a plena adoção da tecnologia pelo mercado. Como solução, é proposto um sistema denominado T-NOVA, um *framework* baseado em *software* livre para monitoramento de NFV que, através de técnicas estatísticas e classificadores Bayesianos, possui foco específico em detecção de anomalias em um ambiente de redes heterogêneo e complexo. A funcionalidade básica consiste em coletar métricas de funcionamento das VNFs a serem monitoradas, bem como dos controladores de infraestrutura virtual, como o OpenStack (OPENSTACK, 2017) e o OpenDaylight (OPENDAYLIGHT, 2017). Através dos resultados, é possível observar que a solução proposta possui um desempenho satis-

fatório para detectar *outliers* da métrica de desempenho quando avaliando a utilização de processamento e consumo de memória. Em alguns casos, esses *outliers* podem indicar uma falha da VNF.

4.2 Trabalhos sobre contêineres em NFV

Esta seção apresenta trabalhos que de alguma maneira utilizam contêineres na solução proposta.

Em (BONDAN; SANTOS; GRANVILLE, 2016) são analisadas três soluções de virtualização baseadas em contêineres indicadas para implementação de funções virtuais de redes, são elas o ClickOS (MARTINS et al., 2014), o CoreOS (COREOS, 2017) e o OSv (KIVITY et al., 2014). De acordo com os autores, o principal motivo para a avaliação de desempenho dessas soluções está no fato de que, apesar de existirem diversas opções e formas de se implementar NFV, muitas delas não estão prontas ou otimizadas para essa finalidade. Outra motivação adicional é que, por serem soluções livres, elas estão disponíveis para utilização a qualquer tempo e necessitam ser avaliadas e experimentadas. Uma vez selecionadas as tecnologias a serem avaliadas, a avaliação de desempenho é focada em métricas de gerenciamento das funções virtuais de rede como, por exemplo, o tempo necessário para executar operações do tipo criar e inicializar uma nova VNF em contêineres e em máquinas virtuais, bem como a utilização de memória e o tráfego de dados nas *interfaces* de redes virtuais. Nos resultados obtidos, os autores colocam as soluções ClickOS e CoreOS como as melhores soluções para NFV, devido à melhor eficiência nos tempos de inicialização da instância virtual, aos melhores tempos de resposta nos tráfegos de rede e ao menor consumo de memória. No caso desta dissertação, o foco primário é na avaliação de desempenho de soluções de virtualização e de conteinerização que sejam genéricas em sua aplicação, independentes de orquestração ou imposições do *framework* padronizado pela ETSI. Soluções como o Docker e o KVM são acessíveis a um maior número de utilizadores, não necessitando de conhecimentos específicos ou habilidades diferenciadas para ClickOS ou CoreOS, por exemplo.

(HEIDEKER; KAMIENSKI, 2016) apresentam a NFV e o SDN (*Software Defined Network*) como possíveis soluções a desafios, como flexibilidade e qualidade de serviço, em tecnologias para Internet das Coisas (*Internet of Things* - IoT) e Cidades Inteligentes (*Smart Cities*). No trabalho é proposta uma solução para gerenciamento de infraestrutura de acesso público à Internet através de WiFi (*Wireless Fidelity*), que utiliza funções virtualizadas de rede como alternativas ao sistemas tradicionais implementados em *appliances* de *hardware* específicos e com custo de manutenção elevado. Especificamente, é apresentada uma solução para o gerenciamento de NAT (*Network Address Translation*) em um ambiente de praças digitais por onde milhares de usuários acessam a Internet

todos os dias. As funções virtualizadas de rede são criadas e destruídas de acordo coma demanda necessária. Para isso, é realizado um gerenciamento elástico dinâmico, similar ao utilizado para gerenciamento e provisionamento de recursos em ambiente de nuvem computacional (*cloud computing*). Através dos resultados é possível confirmar as vantagens na adoção do NFV conforme proposta pelo ETSI mostrando que o desempenho pode ser semelhante ao das soluções tradicionais de *hardware*. As avaliações são produzidas utilizando o KVM e o Xen como plataformas de virtualização tradicionais e contêineres LXC como plataforma de virtualização leve. Em cada caso, é demonstrada a influência de cada plataforma no resultado final e os prós e contras na escolha da tecnologia a ser implementada. Foram utilizadas como métricas, a vazão média de transmissão, o número de requisições admitidos por cada sistema, taxa de transferência, ocupação média de cada VNF e outras. 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.

Em (EIRAS; COUTO; RUBINSTEIN, 2016) é realizada uma avaliação de desempenho de um único *proxy* HTTP implementado como uma VNF, utilizando uma máquina virtual hospedada no KVM ou um contêiner do Docker. Como métrica de comparação, é utilizada a média do tempo total de processamento de 10000 requisições HTTP. São utilizados clientes HTTP enviando as requisições através de uma única VNF de *proxy* Squid 3 para um determinado serviço HTTP responsável por receber e responder essas requisições. 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 a máquina virtual no KVM. Os resultados publicados são uma versão preliminar dos resultados aqui explorados, e uma vez que indicam a viabilidade do uso de um contêiner para implementação de um *proxy* HTTP como VNF, nesse estudo é avaliado o desempenho de diversas instâncias de *proxy* sendo executadas ao mesmo tempo para o processamento de requisições. 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). Além disso, este trabalho avalia o desempenho de máquinas para-virtualizadas no 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.

No trabalho (CZIVA et al., 2015) é apresentado um *framework* para criar, instanciar e gerenciar funções virtuais de rede em uma topologia de rede com OpenFlow (FOUNDATION, 2017) habilitado. Nos cenários avaliados, é explorada a utilização de contêineres visando o menor *overhead* possível no ambiente, com uma instanciação rápida e aceitável índice de reusabilidade do contêiner. Através da utilização de SDN, as requisições e tráfego de rede das VNFs podem ser identificadas por alguma regra, para que seja possível a aplicação de qualidade do serviço e a exibição de eventuais notificações sobre o estado

e a integridade da função de rede instanciada. Os experimentos mostram que operações comuns de um sistema operacional possui ganho de aproximadamente 68% quando utilizando contêineres em relação à implementação utilizando virtualização tradicional. Testes de escalabilidade também mostram que a instanciação de uma VNF utilizando contêineres resulta em poucos milissegundos de latência entre o momento da instanciação e a operação *online*.

4.3 Trabalhos genéricos sobre desempenho de plataformas de virtualização e sistemas de *proxy* HTTP

Diversos trabalhos avaliam o desempenho de plataformas tradicionais de virtualização sem um foco especifico em NFV. No entanto, tais estudos são importantes para elencar diferenças e peculiaridades de cada plataforma, que podem impactar no desempenho quando essas são implementadas para suportar funções virtualizadas de redes.

(YANG; LAN, 2015) propõem um modelo de avaliação de desempenho de máquinas virtuais no KVM. Primeiramente, são apresentadas métricas de avaliação de desempenho para servidores virtuais e então essas métricas são aplicadas para avaliar o desempenho do KVM. A partir disso, é proposto um modelo de avaliação baseado em um modelo de QoS (*Quality of Service)*, que utiliza enfileiramento e busca calcular o desempenho de uma instância virtual utilizando dados obtidos durante a utilização do servidor virtual. Segundo os autores, esse método impede que parâmetros complicados oriundos de um teste de *benchmarking* possam interferir no resultado final, melhorando a eficiência e confiança dos testes em geral. O método proposto consegue avaliar, de forma separada, o desempenho de diferentes partes da instância virtual e é muito útil para provedores de serviços identificarem gargalos no ambiente implementado. Por fim, é fornecida uma discussão que ilustra a utilização do modelo em uma instância controlada do KVM, bem como a acurácia do modelo.

(DUA; RAJA; KAKADIA, 2014) apresentam uma análise comparativa de soluções baseadas em hipervisores e contêineres sob a ótica de PaaS (*Platform-as-a-Service).* A comparação é justificada uma vez que diversos provedores de serviços oferecem soluções baseadas em contêineres a fim de reduzir o *overhead* provocado por máquinas virtuais tradicionais. São exploradas características como isolamento, armazenamento de dados, segurança, tempo de inicialização e comunicação de rede. O foco principal está nos contêineres e em sua capacidade de hospedar aplicações. Diferentes produtos conhecidos no mercado foram avaliados, tais como Docker, Warden Container, lmctfy e OpenVZ. É comparado como cada produto implementa a utilização de processos, sistema de arquivos e isolamento. É verificado também de que forma cada tecnologia de contêineres se beneficia dos recursos oferecidos pelo *kernel* do sistema operacional hospedeiro, ou seja,

do Linux nativo. Ao final, é fornecida uma discussão de aspectos que podem influenciar na escolha da solução de contêiner a ser utilizada e também discussões sobre algumas características ausentes em implementações de contêineres quando aplicadas em PaaS.

(FELTER et al., 2015) avaliam o desempenho do MySQL (MYSQL, 2017), uma aplicação de banco de dados relacional, e do Redis (REDIS, 2017), uma aplicação de bancos de dados em memória (*in-memory*). Ambas são executadas no KVM como virtualização tradicional, no Docker como virtualização leve e também no sistema Linux nativo para servir de linha de base de comparação. O objetivo principal do trabalho é avaliar o *overhead* imposto pelas duas arquiteturas mencionadas. Para atingir o objetivo, são avaliadas métricas importantes relacionadas ao desempenho de aplicações de bancos de dados, como utilização de memória, processamento, uso de rede e taxa de entrada e saída. Nos resultados apresentados, é relatada uma melhora no desempenho do serviço quando se utiliza virtualização por contêiner. Além do supracitado, são fornecidas contribuições quanto aos problemas de desempenho oferecidos pelas tecnologias avaliadas.

Em relação a *proxies* HTTP, (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* habilitado. São avaliadas métricas como latência e vazão em uma rede CAN (*Campus Area Network*), focando somente casos que podem gerar um mau comportamento do cache ou falhas de serviços. De posse das informações, o trabalho contribui de duas formas, sendo a primeira a avaliação do impacto no sistema de *cache* provocado pela transferência de alto volume de dados. Foram coletados dados reais de utilização de dois sistemas de *proxy* submetidos aos testes, um localizado no Japão e outro localizado na Coreia do Sul, interligados através de uma infraestrutura de rede de alta velocidade. Nesse caso, os resultados mostram que objetos concorrentes com tamanho elevado causam uma degradação no sistema de *proxy* é considerável, mesmo para uma quantidade pequena de computadores realizando a transferência. A transferência de objetos com tamanho grande, apesar de representar 0,1% do tráfego total observado, foi responsável por atrasar em 30% o tráfego de objetos menores sendo requisitados por clientes. No pior caso, os tempos de respostas foram incrementados em milhares de milissegundos, o que é caracterizado como um gargalo no sistema. A segunda contribuição corresponde à proposta de uma solução de *cache peer-to-peer* (ponto-a-ponto), que atua de forma colaborativa, compartilhando o esquema de *cache* com mais instâncias para balancear o processamento das requisições. Segundo os autores, a solução proposta consegue reduzir o *overhead* imposto por grandes objetos que entram no sistema de *cache*.

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.

5 AVALIAÇÃO DE DESEMPENHO

Este capítulo avalia o desempenho de *proxies* HTTP em contêineres Docker, em máquinas virtuais KVM e em uma máquina física utilizada como linha de base para comparação. Em relação às máquinas físicas, o ambiente de testes é composto por três máquinas. Conforme ilustrado na Figura 8, uma máquina foi utilizada como cliente, uma máquina foi utilizada como servidor HTTP e a terceira máquina foi utilizada como provedora de serviço de *proxy* HTTP.

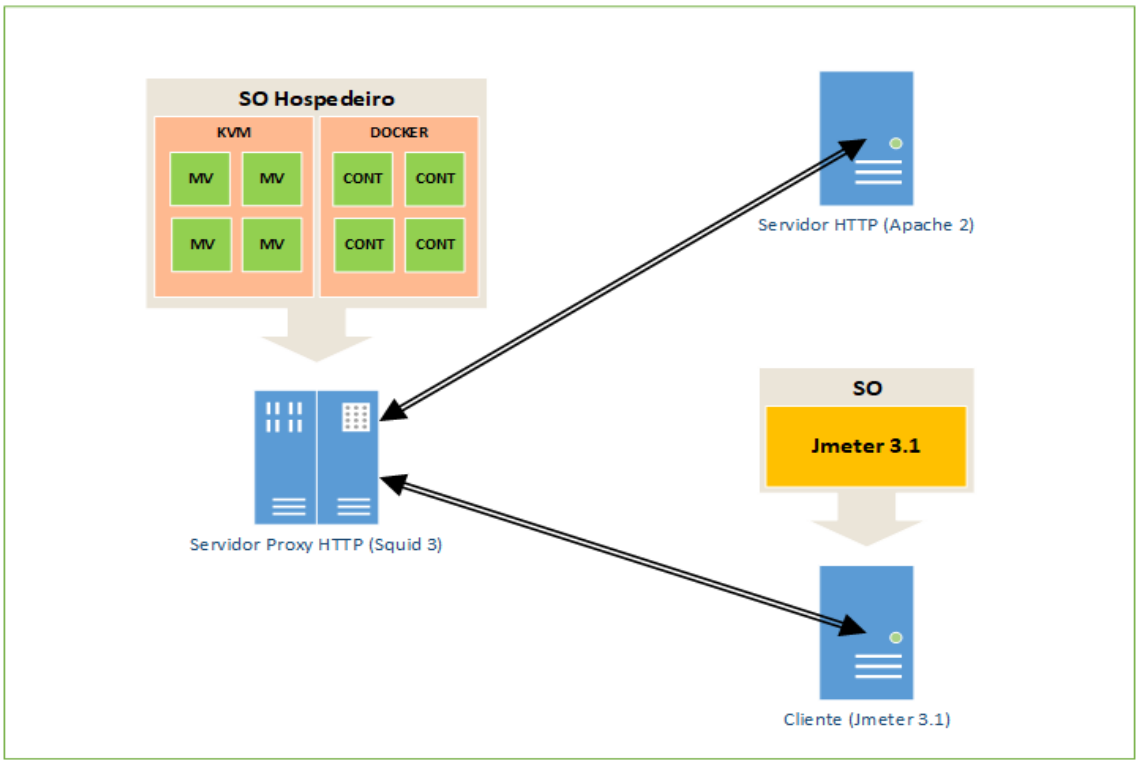
Na máquina cliente, a requisição pode ser realizada pela própria máquina física ou por contêineres que são utilizados como clientes. Na máquina que recebe as requisições, um servidor HTTP é configurado para receber as requisições HTTP dos clientes. No servidor *proxy* HTTP, o Squid 3 (SQUID, 2017) é configurado para interceptar as requisições HTTP do cliente, processá-las e, em seguida fazer, o encaminhamento ao servidor HTTP. O serviço de *proxy* é configurado em uma máquina virtual KVM, em um contêiner Docker ou diretamente na própria máquina física.

Conforme relatado anteriormente, um *proxy* pode ser uma máquina física no caso do Linux nativo instalado diretamente sobre o *hardware*, 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 com *drivers VirtIO*, caracterizando uma implementação para-virtualizada. 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.

Em todos os cenários avaliados, as três máquinas HP Proliant DL160 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 nos 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, 2017b) 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 (APACHE, 2017a), enquanto os

Figura 8 - Cenário utilizado na avaliação de desempenho.



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, por exemplo, Microsoft ForeFront (MICROSOFT, 2017a), Nginx (NGINX, 2017), Untangle (UNTANGLE, 2017), dentre outros. No entanto, optou-se por escolher o Squid 3 por ser de código aberto e possuir grande aceitação pela comunidade desenvolvedora e também pela indústria de telecomunicações. O tamanho do objeto a ser transferido entre o cliente HTTP e o servidor de destino varia de 1 até 1024 kbytes na primeira fase da medição, quando foi utilizado apenas um cliente fazendo a requisições ao servidor. Nas medições seguintes, com múltiplos clientes, optou-se por utilizar um tamanho de objeto fixado em 1024 kbytes, uma vez que o comportamento é similar ao obtido nos demais cenários com tamanhos de objetos menores. Outro ponto que corrobora com essa decisão é o estudo (LLC, 2014), que relata uma média de tamanho de objetos que são processados por serviços HTTP no mundo. Esse estudo relata que em 2014, pela primeira vez, objetos transferidos por serviços HTTP ultrapassaram a média de 1600 kbytes de tamanho.

É importante frisar ainda 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 a mesma configuração do KVM; ou seja, utilizando quatro núcleos de processador e um limite de 4 GB de memória RAM. Esse limite é determinado já que, diferente do que ocorre na virtualização tradicional, não existe reserva de recurso para contêineres; ou seja, eles podem usar a quantidade de

memória disponível pelo sistema operacional hospedeiro até esse limite, que é estipulado no momento da criação do contêiner.

Os experimentos são divididos em quatro partes. A primeira parte apresenta o tempo de instanciação de soluções baseadas em máquinas virtuais tradicionais e contêineres, ou seja, quanto tempo é necessário para inicializar um contêiner versus uma máquina virtual com serviço de *proxy* já configurado. A segunda parte apresenta uma análise na qual um cliente envia requisições HTTP para um determinado servidor. Essas requisições variam no tamanho do objeto a ser transferido e também quanto ao modo de implementação do *proxy* Squid 3 com sistema de *cache* habilitado ou desabilitado. Na terceira parte, é apresentada uma avaliação na qual 10 clientes HTTP fazem requisições simultâneas para até dois *proxies* configurados em contêineres e também em máquinas virtuais tradicionais. Assim como no cenário anterior, o *proxy* pode ter sistema de *cache* habilitado ou não. Por último, e estendendo os resultados da terceira 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 função do melhor desempenho dos contêineres na parte anterior. Em todos os casos das partes 2 a 4, a métrica utilizada é o tempo médio necessário para processar cada requisição HTTP enviada por um ou mais clientes para o servidor alvo, através de instâncias de *proxies* determinadas. Para um objeto, isso corresponde ao tempo decorrido desde o envio da requisição até o recebimento da resposta.

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 das figuras, 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.

5.1 Tempo Médio para Instanciar Contêineres e Máquinas Virtuais

O processo de instanciar uma máquina virtual ou um contêiner deve ser realizado o mais rapidamente possível, uma vez que VNFs são instanciadas muitas vezes à medida que a demanda por recursos começa a impactar o ambiente. É esperado, naturalmente, que contêineres tenham um tempo de entrada em serviço menor, uma vez que não é necessária alocação de recursos em memória, nem utilização de disco rígido virtual. No caso de

Tabela 2 - Tempo médio [s] para instanciar uma VNF por plataforma.

Plataforma	Tempo para instanciar uma VNF (s)
KVM	52 ± 12,05
Docker	4 ± 0,15

máquinas virtuais tradicionais, ainda é necessário que o sistema operacional hospedeiro através do hipervisor saiba exatamente quando ele poderá oferecer recursos de *hardware* adequados ao novo serviço que está sendo instanciado impondo o menor *overhead* possível no ambiente. Independentemente do tipo de plataforma utilizada para suportar VNFs, é conhecido o problema da alocação de VNFs em um ambiente de rede distribuído, que vêm sendo estudado e discutido na literatura (CLAYMAN et al., 2014; MOENS; TURCK, 2014b). Dessa forma, cabe salientar que não somente o tempo de instanciação é relevante; outros fatores também podem determinar a escolha da plataforma de virtualização a ser utilizada como, por exemplo, o uso de recursos de *hardware* durante o processo de instanciação, o tempo de *downtime* durante o processo de configuração e convergência do serviço, o isolamento das VNFs, dentre outras. Para equalizar essas variáveis, algumas estratégias podem ser adotadas como, por exemplo, a utilização de algoritmos ou soluções de reconfiguração de VNFs (OLTEANU; RAICIU, 2012).

Para este experimento, são instanciadas 10 máquinas virtuais e 10 contêineres, sendo o processo repetido 10 vezes para cálculo da média e do intervalo de confiança de 95%. Todas as medições do KVM são realizadas considerando que uma determinada imagem de disco para a máquina virtual já está pronta para ser utilizada.

A Tabela 2 apresenta o tempo médio para instanciar VNFs nas plataformas estudadas. Naturalmente, e devido à própria concepção de soluções baseadas em contêineres, é esperado que o Docker tenha um tempo de instanciação menor. Nesse caso, o tempo para o KVM é 13 vezes maior, o que é um valor considerável, considerando funções de rede e sua criticidade para o ambiente em que opera. Devido ao fato de que cada máquina virtual necessita ter seu próprio disco rígido virtual, uma cópia desse disco é criada para cada máquina virtual instanciada, enquanto na plataforma de contêiner o disco rígido é compartilhado com o sistema operacional hospedeiro. Por menor que seja o tamanho do disco rígido virtual de uma VNF instanciada em uma plataforma de virtualização completa, um pequeno *overhead* é gerado no processo de cópia do disco virtual da nova máquina virtual recém configurada, aumentando o tempo que ela necessita para entrar em operação.

No Apêndice A encontram-se os códigos utilizados nas medições de tempo para instanciar máquinas virtuais e contêineres.

5.2 Cenários com um Cliente e uma Instância de *Proxy*

O objetivo das análises desta seção é oferecer um parâmetro de comparação inicial entre VNFs de *proxy* implementadas através de virtualização tradicional ou conteinerização. Para tal, são configurados serviços de *proxy* HTTP em uma máquina virtual no KVM e em um contêiner do Docker, em duas implementações diferentes, uma com sistema de *cache* habilitado e outra com sistema de *cache* desabilitado. São calculados os tempos médios de processamento de uma requisição HTTP para diferentes tamanho de objetos transferidos: 1, 10, 100 e 1024 kbytes. Junto com as médias de tempo de processamento de uma requisição HTTP, o intervalo de confiança de 95% também é apresentado.

5.2.1 Tempo Médio de Processamento com *Cache* Desabilitado

Nesta seção é feita uma análise da média de tempo de processamento por requisição quando um cliente envia 1000 requisições para um servidor no cenário da Figura 8. Nesse cenário, o sistema de *cache* do Squid 3 está desabilitado. Existem diferentes situações nas quais a implementação de um *proxy* HTTP sem *cache* pode ser interessante como, por exemplo, quando a VNF opera em alguma função de filtro de acesso na arquitetura. Nesse tipo de implementação, o *proxy* é utilizado como um autorizador de tráfego HTTP, verificando a cada requisição se o determinado cliente pode ou não acessar determinado conteúdo.

O KVM é implantado de duas formas, sendo a primeira utilizando virtualização completa e a segunda utilizando para-virtualização através de *drivers* VirtIO (VIRTIO, 2017). O Docker também é implantado de duas formas. A primeira, utilizando o método padrão e convencional que é habilitado no momento da instalação que utiliza NAT para possibilitar a comunicação do contêiner com o mundo externo. A segunda forma é utilizando roteamento estático; nesse caso, o tráfego é enviado para dentro do contêiner através de roteamento, sem o uso do NAT.

Os resultados de tempo médio de processamento, apresentados na Tabela 3, mostram que o Docker tem desempenho próximo ao do sistema Linux nativo, utilizado como linha de base para comparação. Assim, também é possível notar que o Docker possui um desempenho superior ao KVM e que a diferença de desempenho aumenta à medida que cresce o tamanho do objeto transferido. Além disso, apesar de o uso da para-virtualização do KVM melhorar o desempenho em relação à virtualização completa, essa melhoria não é suficiente para superar o desempenho do Docker, independentemente do modo de implantação.

Inicialmente, os resultados superiores do Docker em relação ao KVM podem ser justificados pela própria concepção da arquitetura utilizada por ambas soluções. O Doc-

Tabela 3 - Tempo médio de processamento [ms] por requisição HTTP para um cliente e uma instância de *proxy*, com sistema de cache desabilitado.

Plataforma Avaliada para Virtualização	Objeto de 1 kbyte	Objeto de 10 kbytes	Objeto de 100 kbytes	Objeto de 1024 kbytes
Linux Nativo	1,82 ± 0,01	2,04 ± 0,01	2,64 ± 0,01	11,23 ± 0,01
Docker - NAT	1,96 ± 0,02	2,06 ± 0,02	2,73 ± 0,02	12,65 ± 0,03
Docker - Roteado	1,94 ± 0,02	2,06 ± 0,02	2,73 ± 0,03	12,51 ± 0,03
KVM - Virt. Completa	2,88 ± 0,09	3,98 ± 0,13	5,25 ± 0,09	41,28 ± 0,27
KVM - Para-Virt.	2,37 ± 0,09	3,91 ± 0,08	4,11 ± 0,12	29,48 ± 0,22

ker compartilha o *kernel* e os *drivers* de rede com o sistema operacional hospedeiro. Assim, possui vantagem quando comparado ao KVM, uma vez que não implementa a camada de hipervisor, responsável por grande parte do *overhead* gerado na virtualização. Consequentemente, não necessita traduzir instruções entre a máquina virtual e o sistema operacional hospedeiro. Toda a comunicação entre o sistema operacional hospedeiro e os contêineres é realizado através de *sockets*. Quando o Docker é comparado com o Linux nativo, é possível perceber uma variação nos tempos que é imposta pelo *overhead* gerado no isolamento de rede do Docker. O gerenciador de contêineres do Docker cria uma rede isolada para alocar os contêineres criados sobre ele, de forma que todo o tráfego da rede externa necessita ser encaminhado de alguma forma para dentro do contêiner. Essa rede isolada criada pelo gerenciador do Docker é então acoplada por uma *bridge* em alguma *interface* de rede física.

Mesmo com a imposição de *overhead* ocasionado pelo isolamento de rede do Docker, esse *overhead* é ainda menor quando comparado com os *overheads* para máquinas virtuais alocadas no KVM.

5.2.2 Tempo Médio de Processamento com *Cache* Habilitado

Implementações de *proxy* HTTP geralmente são acompanhadas de configurações de algum tipo de *cache* para diminuir os tempos de processamento de requisições. O Squid 3 possui um sistema de *cache* embarcado muito utilizado em implementações desse tipo de serviço na indústria de telecomunicações. Além de melhorar a experiência do usuário, reduzindo a latência nas navegações na Internet, o *cache* também é utilizado como recurso para economizar banda de transferência no enlace de saída.

No início da década de 2000, devido ao crescimento das tecnologias de transferências de dados, sistemas de *cache* tiveram sua utilização suprimida em detrimento de altas velocidades de navegação providas por sistemas de banda larga por assinatura. Esse comportamento permaneceu por alguns anos até a chegada dos *smartphones* e da

Internet das Coisas (IoT), que demandam cada vez mais recursos de interconectividade e disponibilidade. Com isso, sistemas de *cache* voltaram a ter seu protagonismo entre administradores de sistemas, com o objetivo de limitar e economizar parte da banda que é utilizada por dispositivos que tem o princípio de estarem sempre it online (TECHGENIX, 2014).

De posse dessas informações, esta seção avalia o desempenho de uma VNF implementada como serviço de *proxy* com *cache* habilitado em um cenário em que um cliente faz requisições HTTP para um servidor. A intenção é verificar o quanto um sistema de *cache* no *proxy* pode diminuir os tempos de processamento de requisições e alterar a comparação entre as tecnologias de virtualização. Para realizar essa análise, o sistema de *cache* do Squid 3 foi habilitado no seu arquivo de configuração (`squid.conf`). É importante ressaltar outros parâmetros que são configurados de acordo com a documentação do Squid 3. O primeiro, denominado (*maximum object size*), trata do tamanho máximo de um objeto a ser armazenado na memória do *cache*. Para evitar problemas de desempenho, a documentação recomenda que esse parâmetro não ultrapasse o valor padrão de 1024 kbytes. Assim, o trabalho se restringiu a transferências de objetos de até 1024 kbytes de tamanho. O segundo parâmetro está relacionado à reserva de memória física destinada ao sistema de *cache*. O valor padrão e recomendado foi configurado em 260 Mbytes.

A Tabela 4 mostra que, com o sistema de *cache* habilitado, os tempos diminuem em relação aos obtidos sem *cache* na Tabela 3, conforme esperado. O Docker apresenta, assim como na Seção 5.2.1, desempenho bem próximo do Linux nativo. Caso o ambiente não seja extremamente crítico e com alto volume de tráfego e requisições, a queda de desempenho provocada pelo *overhead* do Docker dificilmente causaria degradação na experiência do usuário.

Outro ponto importante corresponde ao desempenho do KVM quando utiliza-se *drivers* para-virtualizados. Os tempos de processamento da solução para-virtualizada são próximos ao Docker na versão roteada, sendo cerca de 3 ms maiores para o pior caso de objetos de 1024 kbytes. Essa é uma observação importante pois, considerando que uma máquina virtual configurada no KVM possui um isolamento de recursos mais eficiente em relação ao Docker e que uma determinada perda de desempenho possa ser suportada, o KVM passa a ser uma opção a ser considerada. Considerando ainda o isolamento, é importante ressaltar que o Docker compartilha a memória física com o sistema operacional hospedeiro, o que pode gerar problemas de estabilidade do ambiente caso esses sistemas de *cache* não sejam corretamente dimensionados. Já no KVM, apenas a instancia virtual em questão seria afetada no caso de mau comportamento do *cache*.

Tabela 4 - Tempo médio de processamento [ms] por requisição HTTP para um cliente e uma instância de *proxy*, com sistema de cache habilitado.

Plataforma Avaliada para Virtualização	Objeto de 1 kbyte	Objeto de 10 kbytes	Objeto de 100 kbytes	Objeto de 1024 kbytes
Linux Nativo	1,59 ± 0,01	1,79 ± 0,01	1,83 ± 0,01	9,13 ± 0,01
Docker - NAT	1,66 ± 0,02	1,81 ± 0,01	2,43 ± 0,02	10,71 ± 0,03
Docker - Roteado	1,66 ± 0,02	1,81 ± 0,02	2,38 ± 0,03	10,56 ± 0,03
KVM - Virt. Completa	2,12 ± 0,05	3,19 ± 0,09	4,01 ± 0,05	28,16 ± 0,35
KVM - Para-Virt.	2,01 ± 0,03	2,88 ± 0,07	3,79 ± 0,07	13,55 ± 0,08

5.3 Cenários com 10 Clientes e até duas Instâncias de *Proxies*

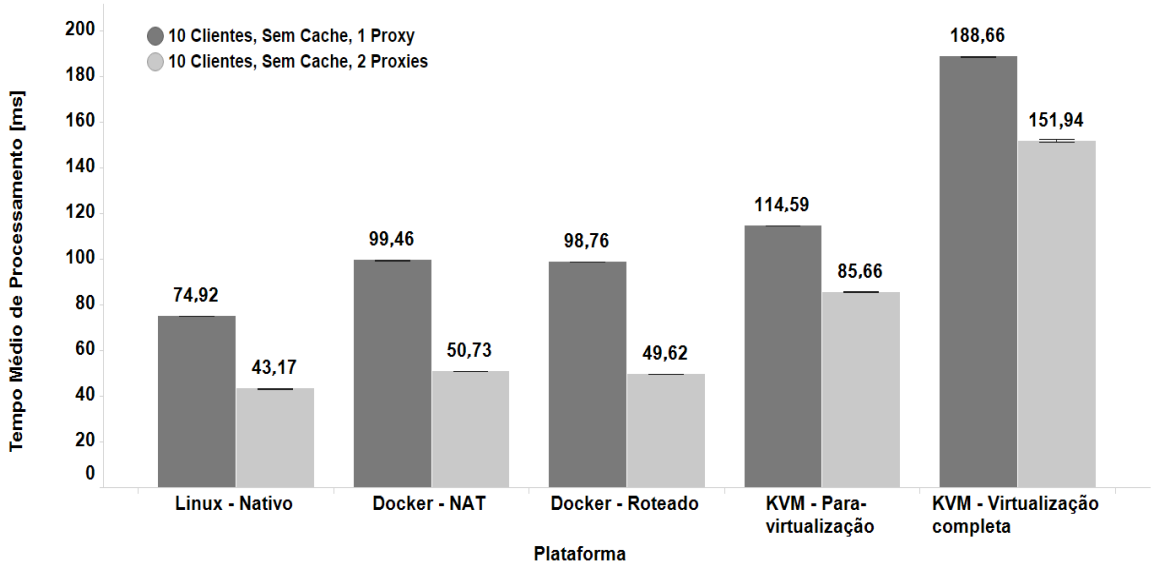
Buscando um melhor entendimento dos resultados obtidos nas Seções 5.2.1 e 5.2.2, nesta seção optou-se por aumentar o número de clientes que realizam requisições HTTP para 10, com o tráfego de requisições passando por uma ou duas instâncias de *proxy*. Para esses cenários, optou-se ainda por fixar o tamanho dos objetos requisitados pelos clientes em 1024 KBytes. O objetivo principal nesse cenário é avaliar se a distribuição de requisições para duas instâncias de *proxy* trabalhando em paralelo pode diminuir o tempo de processamento por requisição, uma vez que requisições poderão ser processadas paralelamente pelas duas VNFs instanciadas.

5.3.1 Tempo Médio de Processamento com *Caches* Desabilitados

Para o cenário descrito anteriormente, nesta seção é avaliado o tempo médio de processamento das requisições HTTP com o sistema de cache do Squid 3 desabilitado. Quando implementadas sobre virtualização, as instâncias são configuradas no KVM ou no Docker. No caso do Linux nativo, dois *sockets* de *proxy* são inicializados com portas diferentes para poder servir de comparativo com as VNFs instanciadas nas tecnologias de virtualização. Naturalmente, nesses cenários os tempos de processamento serão maiores em relação ao cenário anterior com apenas um cliente, já que são 10 clientes requisitando objetos simultaneamente para o servidor HTTP.

A Figura 9 apresenta o tempo médio por requisição para uma ou duas instâncias de *proxy*, sem caches. É possível notar que, novamente, o Docker tem um desempenho próximo ao do Linux nativo, independentemente se encaminha pacotes por roteamento ou NAT para dentro do contêiner. Conforme mencionado na Seção 5.2.1, 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. É

Figura 9 - Tempo médio de processamento por requisição enviada por 10 clientes com uma ou duas instâncias de *proxy* e cache desabilitado.

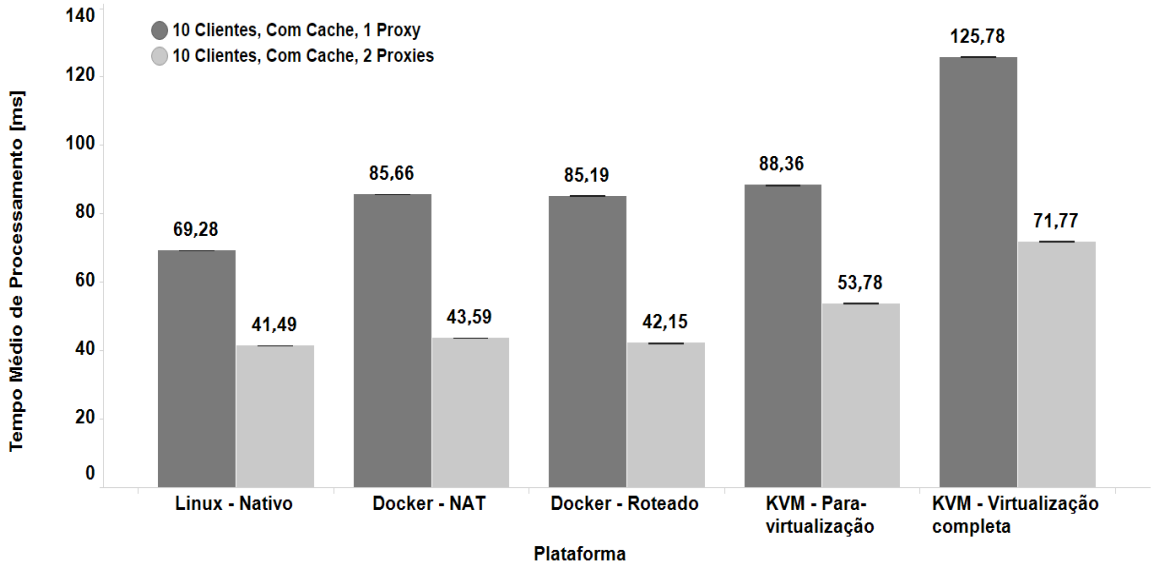


importante salientar que o uso do NAT pode oferecer uma camada de segurança adicional para todas as conexões que entram no contêiner ao custo de um pequeno *overhead*, não permitindo que os contêineres fiquem expostos ao mundo externo.

Observando ainda a Figura 9, o KVM quando utilizando técnicas de para-virtualização consegue melhorar o desempenho em relação à virtualização completa, mas não o suficiente para superar o desempenho do contêiner provido pelo Docker.

Quando as requisições são divididas por mais de um serviço de *proxy* para processamento, 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. Isso é possível pois, ao contrário da virtualização tradicional, não existe alocação prévia de recursos, apenas limites máximos de uso de memória e CPU. No caso do sistema Linux nativo, também é possível perceber que na configuração com dois *sockets* para o processamento da requisições, o desempenho também é melhorado, já que as requisições também são processadas em paralelo.

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



5.3.2 Tempo Médio de Processamento com *Caches* Habilitados

Na sequência em que duas instâncias de *proxy* são utilizadas para processar requisições HTTP de 10 clientes, avalia-se o tempo médio de processamento das requisições HTTP com o sistema de cache do Squid 3 habilitado. Os mesmos parâmetros utilizados na Seção 5.2.2 para habilitar o sistema de *proxy* do Squid 3 são mantidos. Espera-se que nesse cenário novamente 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 Figura 10 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 próximo ao do Linux nativo, e essa proximidade aumenta quando utilizando duas instâncias de *proxy* para processar requisições HTTP dos clientes. Da mesma forma que o observado nos resultados sem *cache*, o desempenho do KVM para-virtualizado e do Docker se aproximam. Entretanto, é importante notar o fato de o hipervisor, nas soluções de virtualização tradicional, separar 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 pode ser importante do ponto de vista do isolamento. 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 de rede virtualizada.

5.4 Análise do Balanceamento de Carga do Docker

Uma vez que o Docker obteve o melhor desempenho entre as tecnologias de virtualização nos experimentos anteriores, esta seção visa analisar em mais detalhes sua capacidade de balanceamento de carga. Para tal, criam-se múltiplos VNFs como proxy, que são responsáveis por tratar requisições de diversos clientes. Cada cliente executa uma instância do Apache JMeter para enviar as requisições, assim como nos experimentos anteriores. Para esse experimento, cada cliente é um contêiner criado no Docker especificamente para essa finalidade. Nesse caso, foram então instanciados 64 contêineres configurados com o Apache JMeter para realizar requisições HTTP simultaneamente para o servidor HTTP Apache 2. Para os clientes, através de testes preliminares, foi possível instanciar no máximo 73 contêineres no ambiente utilizado e que foi apresentado anteriormente na Figura 8. Como procurou-se utilizar um fator de crescimento de clientes em potência de 2 para proceder com os experimentos, optou-se por limitar os clientes no valor máximo de 64.

Para os experimentos, na máquina de *proxy* são criados previamente 32 contêineres executando o Squid 3. Essa quantidade está próxima ao número de contêineres Docker suportados pela máquina utilizada. A partir de 38 instâncias de *proxy*, verificou-se em testes preliminares que os contêineres começam a recusar conexões aleatoriamente, o que pode ser observado através da mensagem "sem rota para o destino". Com os contêineres criados, ativam-se os *proxies*. Varia-se o número de *proxies* entre 1 e 32, sempre em potências de 2. Assim, o tráfego dos 64 clientes é dividido de forma igual entre as instâncias de *proxies* disponíveis para processamento de requisições. Por exemplo, para quatro VNFs de *proxies* disponíveis para utilização, ativam-se quatro contêineres, que recebem tráfego de 16 clientes cada um.

A Figura 11 mostra o tempo médio de processamento em função do número de *proxies* para balancear carga. É possível notar que o balanceamento com duas instâncias ocasiona em uma melhora perceptível no tempo de processamento, de aproximadamente 70 ms, em relação à utilização de apenas um *proxy*. Comportamento semelhante também é observado nos experimentos das Seções 5.3.1 e 5.3.2.

Assim como observado nos experimentos mencionados, a Figura 11 mostra que a versão com NAT habilitado e 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). Devido à carga de operação imposta ao Linux com 64 clientes fazendo requisições nesse cenário, o uso de memória pelas VNFs fez com que o sistema base ficasse saturado e com alta utilização de recursos de *hardware*, piorando os tempos de processamento em relação à versão com NAT quando operando com duas ou mais VNFs.

A Tabela 5 mostra uma comparação de utilização de processador e memória para

Figura 11 - Tempo médio de processamento por requisição enviada por 64 clientes para no máximo 32 instâncias de *proxy* com cache habilitado.

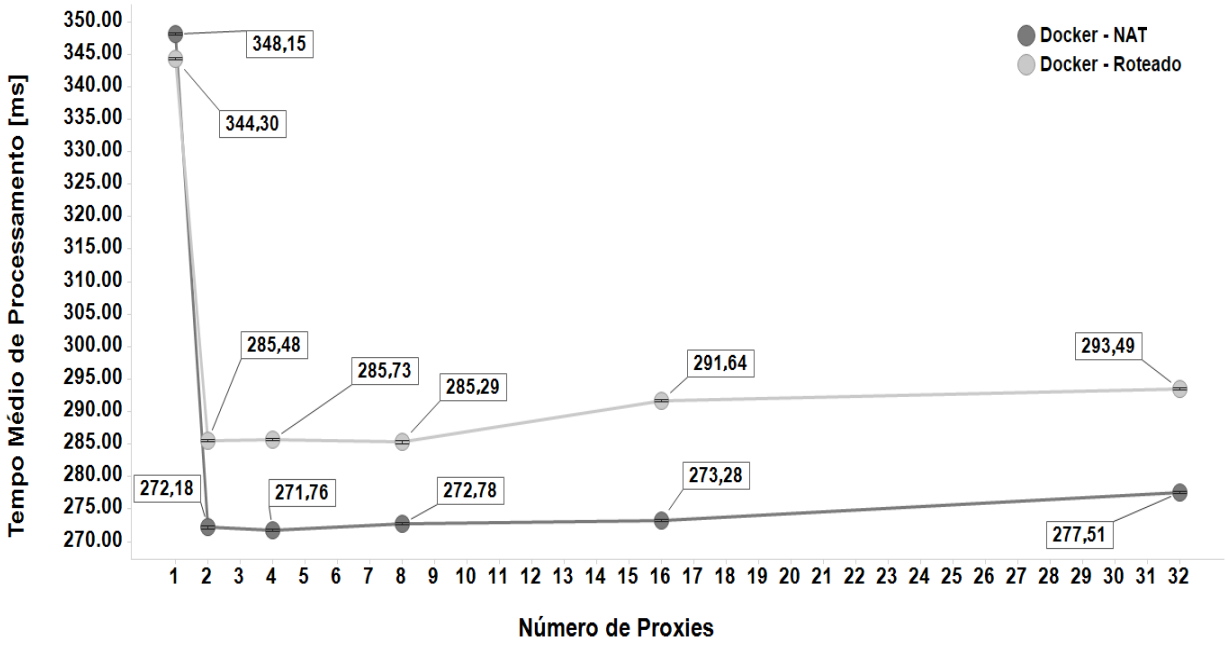


Tabela 5 - Utilização de memória e processador nas versões de VNFs utilizando NAT e Roteamento no Docker.

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

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* maior em relação à versão roteada. É importante frisar que a segurança provida pelo isolamento dos contêineres pode ser um diferencial na segurança das VNFs. 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.

Como a tabela IP é a mesma tanto para os contêineres como para o sistema operacional hospedeiro, o uso de memória tende a aumentar sensivelmente à medida que novas instâncias são adicionadas ao teste. 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 (*swap*). Assim, quando existem *proxies* sendo executados em paralelo, o Docker com roteamento passa a ter desempenho inferior ao Docker com NAT.

Os resultados da Figura 11 também permitem observar que, para mais de dois *proxies* 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 levemente. Isso ocorre devido às características de isolamento do Docker, no qual os processadores e as memórias são compartilhados com todo o sistema e isso pode interferir no desempenho dos contêineres em geral (COMBE; MARTIN; PIETRO, 2016).

6 CONCLUSÕES E TRABALHOS FUTUROS

NFV é um assunto que vem chamando a atenção da indústria de telecomunicações devido à sua proposta de flexibilização e de redução de custos nas implementações de funções de redes. Para substituir com eficiência equipamentos dedicados por soluções virtuais, o desempenho da função de rede virtualizada é um fator extremamente importante e que deve ser cuidadosamente analisado.

Nos experimentos apresentados neste trabalho é possível concluir que contêineres criados na plataforma Docker conseguem ter um desempenho de rede próximo ao do Linux nativo quando aplicados na implantação de *proxies* HTTP. Tal conclusão pode ser explicada pela inexistência de um hipervisor em soluções de conteinerização. Também é possível afirmar que soluções de *proxy* com subsistema de *cache* ativado melhoram significativamente os tempos de processamento de requisições HTTP, em função dos resultados obtidos neste trabalho.

Apesar do desempenho superior do Docker em relação às máquinas virtuais, soluções de virtualização tradicionais, como o KVM, oferecem maior flexibilidade às aplicações NFV. Isso ocorre já que, por exemplo, o sistema operacional convidado pode ser diferente do sistema operacional hospedeiro. Assim, a virtualização tradicional pode ser necessária em alguns cenários. 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 por requisição, 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, uma vez que os resultados mostram uma redução de 50% do tempo de processamento por requisição quando comparado ao da virtualização completa.

Uma breve discussão sobre os tempos de instanciação do contêiner e da máquina virtual também foram fornecidos, mostrando que soluções de *proxy* HTTP construídas sobre contêineres possuem um tempo para entrada em operação extremamente menor, levando praticamente 10% do tempo necessário por uma máquina virtual tradicional. Isso ocorre pois em soluções de contêineres não há a necessidade de se realizar todo o processo de inicialização de um sistema operacional, como ocorre com as máquinas virtuais. Nos contêineres, apenas a aplicação é inicializada e isolada, conferindo assim um tempo de convergência para entrada em serviço menor.

Este trabalho também analisou o quanto o balanceamento de carga pode melhorar o desempenho dos *proxies* virtuais. Em uma primeira avaliação, os resultados mostram que, em todas as soluções de virtualização, instanciar dois *proxies* em paralelo, para distribuir o processamento das requisições, 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.

Analisou-se ainda a capacidade de balanceamento do Docker de acordo com o aumento do número de *proxies* até 32 instâncias. Os resultados mostraram que, para mais de dois *proxies*, a melhora no desempenho não é significativa no cenário considerado. Além disso, ao se aumentar o número de contêineres cresce o consumo de memória, podendo gerar queda de desempenho.

Como recomendação e boas práticas, é importante observar que a para-virtualização no KVM obteve bons tempos de processamento em todos os cenários avaliados, podendo ser uma alternativa interessante à virtualização completa quando implementando NFV. O Docker se mostrou bem versátil na configuração, com boa capacidade de adaptação dos contêineres ao *hardware*, oferecendo condições de limitar recursos garantindo um gerenciamento simples e flexível. Por não possuir um hipervisor em sua arquitetura, se beneficia de chamadas de E/S diretamente ao *hardware* se posicionando bem próximo do Linux nativo.

Para trabalhos futuros, é interessante avaliar a escalabilidade das tecnologias de conteinerização quando implementadas em NFV, inclusive considerando a utilização de soluções que provêm orquestração, gerenciamento e escalabilidade para contêineres, como o Kubernetes (KUBERNETES, 2017). Escalabilidade na prática é importante, pois a solução deve acompanhar o crescimento da demanda de um provedor de serviços. Também é importante avaliar o gerenciamento dessas soluções de conteinerização em um ambiente com alta escalabilidade.

É relevante também avaliar como *frameworks* abertos, tal como o OPNFV (NFV, 2017), podem auxiliar ou melhorar de alguma forma a implantação de VNFs de *proxy* HTTP. Soluções de NFV podem ser melhor gerenciadas com esse tipo de *framework* e tem ocorrido um esforço mundial da indústria no desenvolvimento do mesmo. No entanto, à exceção deste trabalho, não existe na literatura trabalhos que relacionam sistema de *proxy* HTTP como uma função de rede virtualizada fazendo ou não uso de *frameworks* para NFV.

Como os experimentos realizados nesse trabalho foram realizados em um ambiente controlado, não é possível afirmar que o comportamento de desempenho aqui discutido possa ser fielmente representado em qualquer ambiente, por isso, a aplicação dessa metodologia de avaliação em um ambiente real de algum provedor de serviço pode ser interessante para complementar e validar os resultados aqui apresentados. É esperado que, com a evolução da tecnologia de nuvem privada e também de nuvem pública, cada vez mais empresas possam utilizar funções virtuais de redes localizadas externamente à sua organização, por questões de custos e disponibilidade. Dessa forma, uma escolha acertada do tipo de tecnologia a se utilizar pode fazer diferença tanto no desempenho quanto no custo a ser aplicado na manutenção da mesma.

Por fim, é importante também estender este trabalho e avaliar como o isolamento

provido pelo Docker pode afetar a segurança de aplicações NFV. Além disso, uma vez que apenas o Docker como solução de contêiner foi avaliado para utilização como *proxy* HTTP, outras soluções de contêineres podem ser utilizadas para complementar os resultados apresentados neste trabalho.

REFERÊNCIAS

APACHE. *Apache 2 HTTP Server*. 2017. https://projects.apache.org.

APACHE. *JMeter 3.1*. 2017. http://jmeter.apache.org.

BABU, A. et al. System performance evaluation of para virtualization, container virtualization and full virtualization using Xen, OpenVZ and XenServer. In: IEEE. *International Conference on Advances in Computing and Communications*. [S.l.], 2014. p. 247–250.

BARHAM, P. et al. Xen and the art of virtualization. In: ACM. [S.l.]: ACM Symposium on Operating Systems Principles (SOSP), 2003. p. 164–177.

BIEDERMAN, E. W.; NETWORX, L. Multiple instances of the global linux namespaces. In: LINUX SYMPOSIUM. [S.l.]: Linux Symposium, 2006.

BONDAN, L.; SANTOS, C. R. P. dos; GRANVILLE, L. Z. Comparing virtualization solutions for NFV deployment: A network management perspective. In: IEEE. [S.l.]: Symposium on Computers and Communication (ISCC), 2016. p. 669–674.

BUI, T. Analysis of Docker security. *Aalto University School of Science, Finlândia*, abs/1501.02967, 2015. http://arxiv.org/abs/1501.02967.

CINTRA, L. C. *Virtualização com o Xen: Instalando e Configurando o Ambiente*. 2010. Embrapa - Comunicado Técnico, 102.

CITRIX, X. *Citrix XenServer*. 2017. https://www.citrix.com.br/products/xenserver.

CLAYMAN, S. et al. The dynamic placement of virtual network functions. In: IEEE. [S.l.]: Network Operations and Management Symposium (NOMS), 2014. p. 1–9.

COMBE, T.; MARTIN, A.; PIETRO, R. D. To Docker or not to Docker: A security perspective. In: IEEE. [S.l.], 2016. v. 3, n. 5, p. 54–62.

CONDURACHE, C. et al. Performance evaluation of in-kernel system calls. In: IEEE. *Eastern European Regional Conference on the Engineering of Computer Based Systems*. [S.l.], 2015.

CONTAINERS, L. *Linux Containers*. 2017. https://linuxcontainers.org/.

COREOS. *CoreOS: Open Source Projects for Linux Containers*. 2017. https://coreos.com.

COUTO, R. S. et al. Network design requirements for disaster resilience in IaaS clouds. In: . [S.l.: s.n.], 2014. v. 52, n. 10, p. 52–58.

CZIVA, R. et al. Container-based network function virtualization for software-defined networks. In: IEEE. [S.l.]: 20th Symposium on Computer and Communication, 2015.

DATADOG. *8 Surprising Facts About Real Docker Adoption*. 2017. https://www.datadoghq.com/docker-adoption/.

DOCKER. *Docker*. 2017. https://www.docker.com/what-docker.

DOCKER. *Docker Hub*. 2017. https://hub.docker.com/.

DOCKER. *libcontainer*. 2017. https://blog.docker.com/2014/03/docker-0-9-introducing-execution-drivers-and-libcontainer/.

DUA, R.; RAJA, A. R.; KAKADIA, D. Virtualization vs containerization to support PaaS. In: IEEE. *International Conference on Cloud Engineering*. [S.l.], 2014. p. 610–614.

EIRAS, R. S. V.; COUTO, R. S.; RUBINSTEIN, M. G. Performance evaluation of a virtualized HTTP proxy using KVM and Docker. In: IEEE. *International Conference on the Network of the Future (NOF)*. [S.l.], 2016.

ETSI. *ETSI GS NFV 002 V1.1.1: Network Functions Virtualization (NFV); Architectural Framework*. 2013.

ETSI, N. Network functions virtualisation (NFV) architectural framework. In: ETSI. [S.l.], 2013. v. 2, n. 2, p. V1.

FELTER, W. et al. An updated performance comparison of virtual machines and Linux containers. In: IBM. [S.l.], 2015. IBM Research Report - RC25482 (AUS1407-001).

FERNANDES, N. C. et al. Virtual networks: Isolation, performance, and trends. In: . [S.l.: s.n.], 2011. v. 66, n. 5-6, p. 339–355.

FOUNDATION, O. N. *OpenFlow*. 2017. https://www.opennetworking.org/sdn-resources/openflow.

HAN, B. et al. Network function virtualization: Challenges and opportunities for innovations. In: . [S.l.]: IEEE, 2015. v. 53, n. 2, p. 90–97.

HEIDEKER, A.; KAMIENSKI, C. Gerenciamento flexível de infraestrutura de acesso público à Internet com NFV. In: SBRC. *Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos*. [S.l.], 2016.

IBM. Tunning KVM for performance. In: IBM. [S.l.]: IBM Technical Paper, 2nd Ed., 2012.

IDC. KVM: open source virtualization for the enterprise and openstack clouds. In: . [S.l.: s.n.], 2014.

ISHIKAWA, Y. et al. An identification method of PCs behind NAT router with proxy authentication on http communication. In: IEEE. [S.l.]: International Symposium on Applications and the Internet, 2011.

KIM, H.-C. et al. Performance impact of large file transfer on web proxy caching: A case study in a high bandwidth campus network environment. In: IEEE. [S.l.], 2010. v. 52.

KIVITY, A. et al. OSv — optimizing the operating system for virtual machines. In: USENIX. [S.l.]: Annual Technical Conference (USENIX ATC 14), 2014.

KOURTIS, M.-A.; GARDIKIS, G. Statistical-based anomaly detection for NFV services. In: IEEE. [S.l.]: Conference on Network Function Virtualization and Software Defined Networks, 2016.

KUBERNETES. *Kubernetes - Production-Grade Container Orchestration: Automated container deployment, scaling, and management*. 2017. https://kubernetes.io.

KUROSE, J.; ROSS, K. *Computer Networking: a top-down approach*. [S.l.]: Addison-Wesley, 6th ed., 2013.

KVM. *Kernel-based Virtual Machine*. [S.l.]: Open Virtualization Alliance, 2017. https://openvirtualizationalliance.org/what-kvm/overview.

LLC, W. O. *Average Web Page Breaks 1600K*. 2014. http://www.websiteoptimization.com/speed/tweak/average-web-page/.

MARTINS, J. et al. Clickos and the art of network function virtualization. In: USENIX. [S.l.]: Symposium on Networked Systems Design and Implementation (NSDI), 2014.

MAURICIO, L. A. F.; RUBINSTEIN, M. G. Avaliação experimental das plataformas Xen, KVM e openflow no roteamento de pacotes. In: SBC. [S.l.]: XXXIII Simpósio Brasileiro de Telecomunicações, 2015.

MICROSOFT. *Microsoft ForeFront Threat Management Gateway*. 2017. https://www.microsoft.com/en-us/cloud-platform/microsoft-identity-manager.

MICROSOFT. *Microsoft Hyper-V*. 2017. https://msdn.microsoft.com/pt-br/library/hh831531(v=ws.11).aspx.

MIJUMBI, R. et al. Network function virtualization: State-of-the-art and research challenges. In: . [S.l.]: IEEE, 2015. v. 18, n. 1, p. 236–262.

MIJUMBI, R. et al. Network function virtualization: State-of-the-art and research challenges. In: IEEE. [S.l.]: Communications Surveys and Tutorials, 2016. v. 18, n. 1, p. 236—-262.

MOENS, H.; TURCK, F. D. VNF-P: A model for efficient placement of virtualized network functions. In: *10th International Conference on Network and Service Management (CNSM) and Workshop*. [S.l.: s.n.], 2014. p. 418–423.

MOENS, H.; TURCK, F. D. VNF-P: A model for efficient placement of virtualized network functions. In: IEEE. [S.l.]: International Conference on Network and Service Management (CNSM), 2014. p. 418–423.

MYSQL. *MySQL 5.1*. 2017. http://www.mysql.com.

NAKAJIMA, Y.; MASUTANI, H.; TAKAHASHI, H. High-performance vNIC framework for hypervisor-based NFV with userspace vswitch. In: IEEE. [S.l.]: Fourth European Workshop on Software Defined Networks, 2015.

NFV, O. *Open Platform for NFV*. 2017. https://www.opnfv.org/.

NGINX. *NGINX - High Performance Load Balancer, Web Server and Reverse Proxy*. 2017. https://www.nginx.com/.

OLTEANU, V. A.; RAICIU, C. Efficiently migrating stateful middle-boxes. In: ACM. [S.l.]: SIGCOMM Computer Communication Review, 2012. v. 42, p. 93.

OPENDAYLIGHT. *OpenDaylight*. 2017. https://www.opendaylight.org/.

OPENSTACK. *OpenStack Ocata*. 2017. https://www.openstack.org/software/ocata/.

PEETZ, A. *How to vMotion from Intel to AMD - and why not to do it*. 2013. https://www.v-front.de/2013/04/how-to-vmotion-from-intel-to-amd-and.html.

PROJECT, X. *Xen Hypervisor*. 2017. http://xen.org.

QEMU. *QEMU*. 2017. https://www.qemu.org/.

QUEIROZ, G. F. C.; COUTO, R. S.; SZTAJNBERG, A. *TRELIS: Posicionamento de Funções Virtuais de Rede com Economia de Energia e Resiliência*. 2017. 1–14 p.

RAHO, M. et al. KVM, Xen and docker: a performance analysis for ARM based NFV and cloud computing. In: IEEE. [S.l.]: 3rd Workshop on Advances in Information, Electronic and Electrical Engineering, 2015.

REDIS. *Redis 3.2.9*. 2017. http://www.redis.io.

SORIGA, S. G.; BARBULESCU, M. A comparison of the performance and scalability of XEN and KVM hypervisors. In: ICNER. [S.l.]: International Conference on Networking in Education and Research, 2013.

SQUID. *Squid 3 Proxy*. 2017. http://www.squid-cache.org/intro.

STRESS. *Stress for Linux*. 2014. https://people.seas.harvard.edu/ apw/stress.

TECHGENIX. *What's the Buzz on Web Caching?* 2014. http://techgenix.com/whats-buzz-web-caching-part1/.

TSETSE, A. K. et al. A 6to4 gateway with co-located NAT. In: IEEE. [S.l.]: International Conference on Electro/Information Technology (EIT), 2012.

UNTANGLE. *Untangle Gateway*. 2017. http://untangle.com.

VIRTIO. *VirtIO - Paravirtualized Drivers for KVM/Linux*. 2017. http://www.linux-kvm.org/page/virtio.

VIRTUALBOX, O. *Oracle VM VirtualBox*. 2017. https://www.virtualbox.org.

VIRTUOZZO. *OpenVZ Virtuozzo Containers*. 2017. https://openvz.org/.

VMWARE. *VMware ESXi*. 2017. http://www.vmware.com.

WEEK, I. *Containers Commanding More IT Dollars, Survey Finds*. 2016. http://www.informationweek.com/cloud/infrastructure-as-a-service/containers-commanding-more-it-dollars-survey-finds/d/d-id/1325953.

YANG, J.; LAN, Y. A performance evaluation model for virtual servers in KVM-based virtualized system. In: IEEE. [S.l.]: International Conference on Smart City/SocialCom/SustainCom, 2015.

YILE, F. Utilizing the virtualization technology in computer operating system teaching. In: IEEE. [S.l.]: International Conference on Measuring Technology and Mechatronics Automation, 2016.

APÊNDICE A – *Shell Scripts* Utilizados para Instanciar os Conteinêres e as Máquinas Virtuais

A.1 Instanciação dos Contêineres

O *shell script* a seguir foi utilizado para instanciar contêineres no Docker.

```
#!/bin/bash
max=10
for i in ‘seq 1 $max‘
do

let j=$i+1
let porta=21000+$i
cp /home/rodrigo/squid.conf /container/ds-proxy$i/etc/squid.conf
sed -i -- ’s/9002/’$porta’/g’ /container/ds-proxy$i/etc/squid.conf
echo "Criando container ds-proxy’$i’"
start=$SECONDS

    docker run --name ds-proxy$i -h ds-proxy$i -d --publish $porta:$porta
     sameersbn/squid:latest
    docker cp /container/ds-proxy$i/etc/squid.conf
     ds-proxy$i:/etc/squid3/squid.conf
    docker start ds-proxy$i
    docker exec -i ds-proxy$i ifconfig eth0

duration=$(( SECONDS - start ))
echo $duration >> tempo-instanciar-conteiner.txt
done
```

A.2 Instanciação das Máquinas Virtuais

O *shell script* a seguir foi utilizado para instanciar máquinas virtuais no KVM.

```
#!/bin/bash
max=10
for i in ‘seq 1 $max‘
do
```

```
let j=$i+1
let porta=21000+$i
cp /home/rodrigo/squid.conf /vm/ds-proxy$i/etc/squid.conf
sed -i -- 's/9002/'$porta'/g' /vm/ds-proxy$i/etc/squid.conf
echo "Criando VM ds-proxy'$i'"
start=$SECONDS

virt-install \
 -n ds-proxy'$i' \
 --description "VM Proxy '$i'" \
 --os-type=Linux \
 --os-variant=ubuntu \
 --ram=4096 \
 --vcpus=1 \
 --disk path=/var/lib/libvirt/images/ubuntu.img,bus=virtio,size=10 \
 --graphics none \
 --network bridge:br0

cp /vm/ds-proxy$i/etc/squid.conf /kvm/ds-proxy$i/etc/squid/squid.conf
duration=$(( SECONDS - start ))
echo $duration >> tempo-instanciar-vm.txt
done
```

APÊNDICE B – Planos de Teste do JMeter 3.1

Um plano de teste é um roteiro gerado pelo JMeter para realizar a avaliação de desempenho de determinada aplicação. É no plano de teste que as configurações são definidas para executar todo processo de *benchmarking* que esta sendo modelado. Ele agrega as ações a serem executadas e também oferece condições de acesso às repostas da execução, que pode ser uma tabela de dados ou um gráfico para avaliação.

B.1 Exemplo de um Plano de Teste para o Docker Utilizando NAT

```
<?xml version="1.0" encoding="UTF-8"?>
<jmeterTestPlan version="1.2" properties="2.3" jmeter="2.8.20130705">
  <hashTree>
    <TestPlan guiclass="TestPlanGui"
    testclass="TestPlan" testname="ProxyHTTP_UERJ_DOCKER"
    enabled="true">
      <stringProp name="TestPlan.comments"></stringProp>
      <boolProp name="TestPlan.functional_mode">true</boolProp>
      <boolProp name="TestPlan.serialize_threadgroups">true</boolProp>
      <elementProp name="TestPlan.user_defined_variables" elementType="Arguments"
      guiclass="ArgumentsPanel" testclass="Arguments" testname="Variáveis
      Definidas Pelo Usuário" enabled="true">
        <collectionProp name="Arguments.arguments"/>
      </elementProp>
      <stringProp name="TestPlan.user_define_classpath"></stringProp>
    </TestPlan>
    <hashTree>
      <ThreadGroup guiclass="ThreadGroupGui" testclass="ThreadGroup"
      testname="DockerNAT" enabled="true">
        <stringProp name="ThreadGroup.on_sample_error">
        continue</stringProp>
        <elementProp
        name="ThreadGroup.main_controller" elementType="LoopController"
        guiclass="LoopControlPanel" testclass="LoopController"
        testname="Controlador de Iteração"
        enabled="true">
```

```
    <boolProp name="LoopController.continue_forever">
    false</boolProp>
    <stringProp name="LoopController.loops">1000</stringProp>
  </elementProp>
  <stringProp name="ThreadGroup.num_threads">10</stringProp>
  <stringProp name="ThreadGroup.ramp_time">1</stringProp>
  <longProp name="ThreadGroup.start_time">1483621871000</longProp>
  <longProp name="ThreadGroup.end_time">1483621871000</longProp>
  <boolProp name="ThreadGroup.scheduler">false</boolProp>
  <stringProp name="ThreadGroup.duration"></stringProp>
  <stringProp name="ThreadGroup.delay">
  </stringProp>
</ThreadGroup>
<hashTree>
  <LoopController
  guiclass="LoopControlPanel" testclass="LoopController"
  testname="Controlador de Iteração"
  enabled="true">
    <boolProp name="LoopController.continue_forever">true</boolProp>
    <stringProp name="LoopController.loops">5</stringProp>
  </LoopController>
  <hashTree>
    <HTTPSamplerProxy
    guiclass="HttpTestSampleGui" testclass="HTTPSamplerProxy"
    testname="DNT - Requisição HTTP"
    enabled="true">
      <elementProp
      name="HTTPsampler.Arguments" elementType="Arguments"
      guiclass="HTTPArgumentsPanel" testclass="Arguments"
      testname="Variáveis
      Definidas Pelo Usuário"
      enabled="true">
        <collectionProp
        name="Arguments.arguments"/>
      </elementProp>
      <stringProp name="HTTPSampler.domain">10.1.1.1</stringProp>
      <stringProp name="HTTPSampler.port">80</stringProp>
      <stringProp name="HTTPSampler.proxyHost">192.168.0.2</stringProp>
      <stringProp name="HTTPSampler.proxyPort">3128</stringProp>
```

```
            <stringProp name="HTTPSampler.connect_timeout"></stringProp>
            <stringProp name="HTTPSampler.response_timeout"></stringProp>
            <stringProp name="HTTPSampler.protocol"></stringProp>
            <stringProp name="HTTPSampler.contentEncoding"></stringProp>
            <stringProp name="HTTPSampler.path">/teste.1M</stringProp>
            <stringProp name="HTTPSampler.method">GET</stringProp>
            <boolProp name="HTTPSampler.follow_redirects">true</boolProp>
            <boolProp name="HTTPSampler.auto_redirects">false</boolProp>
            <boolProp name="HTTPSampler.use_keepalive">true</boolProp>
            <boolProp name="HTTPSampler.DO_MULTIPART_POST">false</boolProp>
            <boolProp name="HTTPSampler.monitor">false</boolProp>
            <stringProp name="HTTPSampler.embedded_url_re"></stringProp>
          </HTTPSamplerProxy>
          <hashTree/>
        </hashTree>
      </hashTree>
    </hashTree>
  </hashTree>
</jmeterTestPlan>
```

APÊNDICE C – *Scripts* Geradores de Requisições HTTP

C.1 Exemplo de um *script* gerador de requisições HTTP

```
#!/bin/bash
#

max1=32

for i in ‘seq 1 $max1‘
do
let j=$i+1

 cp /home/rodrigo/UERJ_MEDICOES/Parte2/docker/JMETER_PEL_UERJ-DOCKER-CMD.jmx
     /sharedContPEL/ds$i/ds$i.jmx
 sed -i -- ’s/172.17.0.2/172.17.0.2/g’ /sharedContPEL/ds$i/ds$i.jmx
 sed -i -- ’s/3128/21001/g’ /sharedContPEL/ds$i/ds$i.jmx
 docker exec -i ds$i jmeter -Djava.rmi.server.hostname=192.168.220.$j -n -p
    /sharedContPEL/jmeter.properties -t /sharedContPEL/ds$i/ds$i.jmx
    -l /sharedContPEL/64clients-2proxy.csv >> run64-1.txt &

done
```