Desempenho de Rede em Contêineres Docker e Máquinas Virtuais: Um Estudo Quantitativo com iPerf3

João Guilherme Vargas[1], **Mateus Unulino dos Passos**[1],
Matheus Roberto da Silva Corrêa[1], **Raphael Mendonça Bussons da Silva**[1]

[1] Centro de Ciências Tecnológicas - Universidade do Estado de Santa Catarina (UDESC)
R. Paulo Malschitzki, 200 - Zona Industrial Norte, Joinville - SC, 89219-710

jotaguivargas@gmail.com, mateus.passos1605@edu.udesc.br,

matheus.correa@edu.udesc.br, raphaelbussons@gmail.com

Abstract. *This work presents a comparative analysis of network performance between virtual machines (VMs) created in VirtualBox and Docker containers, using the iPerf3 tool to measure bandwidth. This research, of an experimental and quantitative nature, was carried out on machines with Ubuntu 22.04.5 LTS and Ubuntu 24.04, for the containers and VMs respectively, using bridge and host network configurations in the containers and internal network in the VMs.*

Resumo. *Este trabalho apresenta uma análise comparativa do desempenho de rede entre máquinas virtuais (VMs) criadas no VirtualBox e contêineres Docker, utilizando a ferramenta iPerf3 para medir largura de banda. A pesquisa, de natureza experimental e quantitativa, foi conduzida em máquinas com Ubuntu 22.04.5 LTS e Ubuntu 24.04, para os contêineres e VMs respectivamente, usando configurações de rede bridge e host nos contêineres e internal network nas VMs.*

1. Contextualização

A virtualização só ganhou adoção em larga escala nos anos 1990, com o surgimento da VMware [Susnjara and Smalley 2024b]. A abstração proporcionada pelas máquinas virtuais ou *virtual machines* (**VMs**) facilitou o escalonamento de aplicações, especialmente com a popularização do GNU/Linux e de servidores web como o Apache HTTP Server, que demandavam ambientes isolados e flexíveis.

Com a evolução da infraestrutura de internet e a crescente necessidade de desempenho (redução de latência e aumento de velocidade), impulsionada pela era dos *smartphones*, as limitações das VMs tornaram-se evidentes: alto consumo de recursos computacionais e energia. Isso levou ao desenvolvimento de uma alternativa mais eficiente: os contêineres [Muñoz 2019].

Diferentemente das VMs, os contêineres compartilham o sistema operacional do *host*, oferecendo isolamento com menor sobrecarga. Essa inovação foi crucial para a computação em nuvem, permitindo que empresas como Google LLC, Amazon.com Inc. e Microsoft Corporation oferecessem serviços escaláveis, como *streaming*, redes sociais e comércio eletrônico.

2. Objetivo

Este trabalho tem como objetivo principal comparar o desempenho de rede, focado na métrica de largura de banda, entre ambientes de virtualização distintos: VMs criadas com o VirtualBox e contêineres gerenciados pelo Docker. Utilizando a ferramenta iPerf3[1], a análise investiga o impacto de diferentes configurações de rede, avaliando a comunicação com o modo *bridged* nas VMs e com os modos *bridge* e *host* nos contêineres.

3. Revisão da literatura

A literatura acadêmica aponta a vantagem de desempenho dos contêineres em relação às máquinas virtuais. Um estudo seminal de Felter et al. [Felter et al. 2015] demonstrou que contêineres se aproximam do desempenho nativo em métricas de *CPU* e memória, enquanto as VMs apresentam uma sobrecarga consideravelmente maior em todas as categorias avaliadas.

Com foco específico no desempenho de rede, trabalhos como o de Kozhirbayev e Sinnott [Kozhirbayev and Sinnott 2017] aprofundaram a análise, investigando os modos de rede do Docker e concluindo que mesmo a comunicação em *bridge* é mais eficiente que a de VMs. A relevância desses achados é acentuada em arquiteturas de microsserviços, onde a sobrecarga de rede das VMs pode se tornar um gargalo [Gomes et al. 2020].

Esses achados, que destacam a eficiência dos contêineres em comparação com as VMs devido ao menor overhead, são cruciais para a presente pesquisa. Eles justificam a exploração de diferentes configurações de rede em contêineres, como bridge e host, e a análise de seu impacto na largura de banda, aprofundando a compreensão das vantagens práticas para aplicações distribuídas.

4. Ferramentas e Configuração do Ambiente

4.1. VirtualBox

VirtualBox[2] é um programa de virtualização de código aberto desenvolvido atualmente pela Oracle Corporation, que permite criar e executar máquinas virtuais em um computador físico. Assim, o programa emula um computador completo, incluindo sistemas operacionais e aplicativos [Leandro 2025].

4.1.1. Como funciona

O VirtualBox, assim como outras tecnologias de VMs, opera através de um processo chamado virtualização, que permite a criação de um ambiente computacional completo e funcional dentro de um sistema operacional já existente. Essencialmente, uma VM é um ”computador dentro de um computador”[Microsoft Azure 2025]. Isso é possível graças a um *software* fundamental chamado monitor de máquina virtual ou *hypervisor* (**VMM**). O VMM tem a função de criar uma camada de abstração sobre o *hardware* físico da máquina hospedeira (o computador real), como processador, memória RAM, armazenamento e rede.

[1] https://iperf.fr/

[2] https://www.virtualbox.org/

Este *software* de virtualização, no caso o VirtualBox, gerencia e aloca esses recursos de *hardware* para a máquina virtual "convidada". Dessa forma, é possível instalar e executar um sistema operacional completamente diferente (o "*guest OS*") do sistema operacional do computador hospedeiro (o "*host OS*"), como rodar o Windows em um Mac ou o Linux em um Windows [Oracle 2025]. A VM funciona em uma janela isolada do sistema principal, o que garante que qualquer *software* ou alteração dentro dela não afete o sistema hospedeiro, proporcionando um ambiente seguro para testes de *software*, desenvolvimento e execução de aplicações em diferentes plataformas [Red Hat 2025]. O VirtualBox, especificamente, utiliza um *hypervisor* do "Tipo 2", o que significa que ele é instalado como uma aplicação sobre o sistema operacional do hospedeiro já existente [VMware 2025].

Essa distinção, onde o *hypervisor* atua como um *software* sobre o sistema operacional principal, permite uma maior flexibilidade, mas pode introduzir uma leve sobrecarga de desempenho em comparação com os *hypervisors* de Tipo 1, que são executados diretamente no *hardware*.

4.1.2. Pré-requisitos

Segundo o guia do usuário do VirtualBox [3], para instalar o programa, os seguintes requisitos devem ser atendidos:

1. A versão do Ubuntu[4] deve ser uma das seguintes: `20.04`, `22.04`, `24.04` ou `24.10`. Essas versões são as que apresentam suporte oficial da Oracle Corporation.
2. Ter a versão `6.5.3` ou superior do Qt[5] instalado na máquina. Alguns sistemas realizam, automaticamente, a instalação desse pacote quando o VirtualBox é instalado.
3. O GCC[6], o Make do GNU[7] e os *headers* apropriados ao *kernel* sendo utilizado no sistema. Essas ferramentas e arquivos são necessários para a instalação de alguns pacotes do VirtualBox.

4.1.3. Instalação

Existem algumas maneiras diferentes de se instalar o VirtualBox em um sistema. Optou-se por utilizar o repositório "apt"de acordo com o guia do usuário [3].

Configurar o repositório apt do Virtualbox

```
# Atualizar os repositórios
sudo apt update
```

[3] https://download.virtualbox.org/virtualbox/7.1.10/UserManual.pdf
[4] https://ubuntu.com/
[5] https://www.qt.io/
[6] https://gcc.gnu.org/
[7] https://www.gnu.org/software/make/

```
# Instalar requisitos para certificados
sudo apt install -y gpg curl
# Baixar e assinar certificados VirtualBox
curl -fL
 ↪  https://www.virtualbox.org/download/oracle_vbox_2016.asc |
 ↪  sudo gpg --dearmor --yes -o
 ↪  /usr/share/keyrings/oracle-virtualbox-2016.gpg
# Adicionar o repositório na lista
echo \
  "deb [arch=$(dpkg --print-architecture) \
  signed-by=/usr/share/keyrings/oracle-virtualbox-2016.gpg] \
  https://download.virtualbox.org/virtualbox/debian \
  $(. /etc/os-release && echo
   ↪  "${UBUNTU_CODENAME:-$VERSION_CODENAME}") contrib" | \
  sudo tee /etc/apt/sources.list.d/virtualbox.list > /dev/null
# Atualizar os repositórios
sudo apt update
```

Instalar pacote do Virtualbox

```
# Instalar pacotes
sudo apt install virtualbox-7.1
```

Instalar pacotes de extensão do Virtualbox

Embora não seja necessário para o funcionamento do VirtualBox, existe a possibilidade de serem adicionados pacotes de extensão que adicionam funcionalidades para o usuário. É recomendado que o usuário utilize, pelo menos, o pacote de extensão oficial oferecido pela Oracle Corporation. Para instalar esse pacote, podem ser seguidos os seguintes passos:

```
# Instalar requisitos
sudo apt install linux-headers-`uname -r`
# Configurar variavel
VBOX_VERSION=$(VBoxManage --version | sed 's/r.*//')
# Baixar pacote de extensão
wget "https://download.virtualbox.org/virtualbox/${VBOX_VERSION}
 ↪  /Oracle_VirtualBox_Extension_Pack-${VBOX_VERSION}.vbox-extpa
 ↪  ck"
# Adicionar no Virtualbox
sudo modprobe vboxdrv
sudo VBoxManage extpack install --replace "Oracle_VirtualBox_Ext
 ↪  ension_Pack-${VBOX_VERSION}.vbox-extpack"
# Verificar se o pacote de extensão foi adicionado
VBoxManage list extpacks
```

4.1.4. Configurando o ambiente

4.2. Docker

O Docker é uma plataforma de código aberto que permite aos desenvolvedores construir, implementar, executar, atualizar e gerenciar contêineres. Contêineres são componentes executáveis padronizados que combinam o código fonte da aplicação com as bibliotecas e dependências do sistema operacional (SO) necessárias para executar esse código em qualquer ambiente.[Susnjara and Smalley 2024a]
É um ecossistema completo, composto por diversas ferramentas e componentes para o desenvolvimento, empacotamento e execução de aplicações em contêineres. Entre eles estão: *Docker Engine*, *Docker CLI*, *Docker Compose*, *Docker Desktop*, *Docker Hub*, além de ferramentas para desenvolvimento e orquestração, como *Docker BuildKit* e *Docker Swarm*. Neste trabalho, o foco restringe-se ao uso do *Docker Engine*, *Docker CLI* e *Docker Compose*.

4.2.1. Como funciona

O Docker utiliza uma arquitetura cliente-servidor. O cliente Docker conversa com o daemon Docker, que realiza o trabalho pesado de construir, executar e distribuir seus contêineres Docker. O cliente e o daemon Docker podem rodar no mesmo sistema, ou você pode conectar um cliente Docker a um daemon Docker remoto. O cliente e o daemon Docker se comunicam usando uma API REST, por meio de sockets UNIX ou uma interface de rede. Outro cliente Docker é o Docker Compose, que permite trabalhar com aplicações compostas por um conjunto de contêineres.[Docker Inc. 2025a]

A operação eficiente do daemon é, portanto, central para a orquestração e o ciclo de vida dos contêineres Docker, sendo a base para as interações realizadas pelo cliente Docker.

Daemon Docker

O daemon Docker (dockerd) escuta requisições da API do Docker e gerencia objetos do Docker, como imagens, contêineres, redes e volumes. Um daemon também pode se comunicar com outros daemons para gerenciar serviços Docker.[Docker Inc. 2025a]

Cliente Docker

O cliente Docker (docker) é a principal forma com que muitos usuários interagem com o Docker. Quando você usa comandos como docker run, o cliente envia esses comandos para o dockerd, que os executa. O comando docker utiliza a API do Docker. O cliente Docker pode se comunicar com mais de um daemon.[Docker Inc. 2025a]

4.2.2. Pré-requisitos

Para instalar o Docker Engine, você precisa da versão 64 bits de uma das seguintes versões do Ubuntu:

- Ubuntu Oracular 24.10
- Ubuntu Noble 24.04 (LTS)
- Ubuntu Jammy 22.04 (LTS)

O Docker Engine para Ubuntu é compatível com as arquiteturas: x86_64 (ou amd64), armhf, arm64, s390x e ppc64le (ppc64el)[Docker Inc. 2025b].

A conformidade com esses requisitos de sistema operacional e arquitetura é essencial para garantir a compatibilidade e a estabilidade da instalação do Docker Engine, preparando o ambiente para a execução dos testes de desempenho.

4.2.3. Instalação

Existem algumas maneiras diferentes de se instalar o Docker em um sistema. Este estudo usa o repositório "apt"de acordo com a documentação oficial do Docker[8].

Configurar o repositório apt do Docker

```
# Atualizar os repositórios
sudo apt update
# Instalar requisitos para certificados
sudo apt install ca-certificates curl
sudo install -m 0755 -d /etc/apt/keyrings
# Baixar certificados do Docker
sudo curl -fsSL https://download.docker.com/linux/ubuntu/gpg -o
 ↪  /etc/apt/keyrings/docker.asc
# Dar permissão de leitura ao certificado
sudo chmod a+r /etc/apt/keyrings/docker.asc
# Adicionar repositório as fontes do apt validando certificado
echo \
"deb [arch=$(dpkg --print-architecture)
 ↪  signed-by=/etc/apt/keyrings/docker.asc]
 ↪  https://download.docker.com/linux/ubuntu \
$(. /etc/os-release && echo
 ↪  "${UBUNTU_CODENAME:-$VERSION_CODENAME}") stable" | \
sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
# Atualizar os repositórios
sudo apt update
```

[8]https://docs.docker.com/engine/install/ubuntu/#install-using-the-repository

Instalar os pacotes do Docker

```
# Instalar pacotes
sudo apt install docker-ce docker-ce-cli containerd.io
 ↪  docker-buildx-plugin docker-compose-plugin
```

Verificar a instalação

A verificação da instalação é realizada com a execução da imagem 'hello-world':

```
# Verificando
docker run hello-world
```

É esperado que seja retornada uma mensagem explicando os passos que foram necessários para a imagem ser executada

4.2.4. Configurando o ambiente

No que tange aos contêineres, para facilitar a replicação dos testes, foram preparados alguns arquivos pré-configurados em um repositório online[9]. Sendo esses arquivos: "compose-bridge.yml"(Listagem 1) e "compose-host.yml"(Listagem 2) que preparam o agrupamento desses contêineres e suas configurações de redes, "Dockerfile"(Listagem 3) que cuida da configuração base das imagens e os scripts "script_receiver.sh"(Listagem 4), "script_sender.sh"(Listagem 5) e "script_formatter.sh"(Listagem 6) que cuidam de transmitir os dados e formatar os resultados. Ao fim o projeto estará formatado segundo a Figura 1.

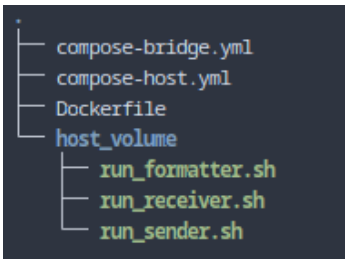

Figura 1. Estrutura do projeto na parte do Docker.

4.3. iPerf3

Para realizar testes de desempenho e largura de banda em redes de computadores, uma das ferramentas mais utilizadas atualmente é o iPerf3. Trata-se de uma aplicação de código aberto desenvolvida pelo Energy Sciences Network (ESnet), amplamente adotada tanto em ambientes acadêmicos quanto corporativos. O iPerf3 permite realizar medições precisas da largura de banda entre dois dispositivos conectados a uma rede, utilizando os protocolos TCP, UDP e SCTP (em sistemas compatíveis).

[9]https://github.com/JoaoVargas/2025-REC-TE/tree/main/Docker

Para cada teste, o iPerf3 fornece métricas detalhadas sobre largura de banda efetiva, taxa de perda de pacotes e outros parâmetros de desempenho.

Instalação nos contêineres

O iPerf3 pode ser incluído já na especificação dos contêineres no Dockerfile como pode ser visto na Listagem 3 e pode-se automatizar a execução dos testes com scripts *bash* nos arquivos para o uso com *Docker Compose* (Listagem 1). Mas pode-se, uma vez dentro dos contêineres (`docker exec -it nome_do_container bash`), 'manualmente' fazer `iperf3 -s` no servidor que a aplicação escutará na porta 5201. No cliente, `iperf3 -c hostname_do_servidor -t n` faz com que o teste seja executado por *n* segundos.

5. Metodologia Experimental

5.1. Introdução

Esta seção tem a pretensão de cobrir os passos necessários para replicar o experimento e obter resultados similares.

5.2. Método

Este trabalho adota uma abordagem experimental para comparar o desempenho de comunicação de rede entre dois tipos de ambientes virtualizados: máquinas virtuais (VMs) e contêineres Docker. A metodologia foi estruturada para permitir a reprodução dos testes, bem como a coleta e análise de dados de forma sistemática.

As ferramentas e tecnologias utilizadas foram o iPerf3, para a medição do desempenho de rede, Docker, para a criação e gerenciamento de contêineres Linux isolados, VirtualBox para a criação das máquinas virtuais, Ubuntu 24.04.2 LTS como sistema operacional base, tanto nas VMs quanto nos contêineres e o jq, para processar dados JSON que foram gerados pelo iPerf3.

5.3. Plano de Testes

Para realizar uma análise comparativa dos dados de vazão gerados pelos scripts em conjunto com o iPerf3, tanto nas VMs como nos contêineres, foi determinado:

- Determinar o escopo do teste a se realizar com iPerf3;
- Escrever um arquivo para uso com *Docker Compose* com configuração de rede *bridge*;
- Escrever outro com configuração de rede *host*;
- Gerar dados estruturados a partir dos testes nos contêineres;
- Gerar dados estruturados a partir dos testes nas VMs;
- Gerar gráficos para a vazão por execução a partir dos dados estruturados - tanto para as VMs como para os contêineres;
- Contrapor os resultados;

5.4. Ambiente de Testes

Os experimentos foram conduzidos em duas máquinas físicas com suporte a virtualização, nos seguintes cenários distintos:

1. Cenário com máquinas virtuais: duas VMs conectadas em uma rede (*bridge*), executando Ubuntu 24.04, com iPerf3 instalado.
2. Primeiro cenário com contêineres Docker: dois contêineres conectados por meio de uma rede *bridge*, iPerf3 instalado bem como utilitários de rede e scripts para automatizar o processo de execução dos testes e coleta de resultados.
3. Segundo cenário com contêineres Docker: dois contêineres conectados por meio de uma rede *host*, iPerf3 instalado bem como utilitários de rede e scripts para automatizar o processo de execução dos testes e coleta de resultados.

5.4.1. Contêiners Docker

Para avaliar o desempenho da comunicação em rede entre contêineres, foram configurados dois contêineres Docker — `sender` e `receiver` — em uma única máquina física. Ambos os contêineres se comunicam por meio de uma rede do tipo *bridge* (e, em um segundo momento,*host*). A orquestração dos serviços foi feita com o *Docker Compose*, utilizando os seguintes componentes principais:

- **Rede *bridge* personalizada**: Os dois contêineres estão conectados à rede `my-network`, definida no arquivo `compose-bridge.yml`, permitindo comunicação direta por IP interno e também por nome do host.
- **Modo *host***: Os dois contêineres comunicam-se indiretamente através da máquina host, como especificado no arquivo `compose-host.yml`.
- **Volumes**: Um volume montado do host (`./host_volume`) é compartilhado entre os contêineres por meio de dois volumes nomeados (`receiver_volume` e `sender_volume`), possibilitando o armazenamento e compartilhamento persistente de arquivos de log.
- **Contexto de *build***: Ambos os serviços são construídos a partir de um único `Dockerfile`, baseado na imagem `debian:12.11`, que instala todas as ferramentas necessárias para os testes de rede, como `iperf3`, `ping`, `net-tools` e `jq`.
- **Comandos de inicialização**:
  - O container `receiver` executa o script `run_receiver.sh`, que inicia o servidor `iperf3` e permanece ativo com um `sleep infinity`.
  - O container `sender` executa o script `run_sender.sh` para se conectar ao receptor e realizar os testes, seguido por `run_formatter.sh` para processar os resultados.
- **Dependência de serviços**: O container `sender` depende explicitamente do `receiver`, garantindo que o servidor esteja ativo antes do início dos testes do cliente.

5.4.2. Máquinas Virtuais

Para avaliar o desempenho da comunicação em rede entre máquinas virtuais (VMs), foram configuradas duas máquinas virtuais utilizando o VirtualBox. Ambas as VMs foram

provisionadas com uma imagem ISO do Ubuntu 24.04.2 e tiveram o utilitário iPerf3 pré-instalado para medições de desempenho. A rede do tipo *bridge* foi configurada diretamente na interface gráfica do VirtualBox, acessando a aba de Configurações de Rede de cada VM.

5.5. Execução

Depois de construir os contêineres (`docker compose -f compose-bridge.yml build` e `docker compose -f compose-host.yml build`), executa-se o comando `docker compose -f compose-bridge up` ou `docker compose -f compose-host up` para instanciar os contêineres. O resultado da execução dos scripts com os testes do iPerf pode ser verificado no diretório `iperf3_receiver_logs`. É importante atentar para o fato de que as execuções devem ser isoladas ou deve-se dar nomes diferentes para os contêineres do teste *bridge* e *host*, caso contrário haverá conflitos de nome.

Para os testes de desempenho de rede entre as VMs com Ubuntu 24.04, o processo é o seguinte:

Inicie ambas as VMs pelo VirtualBox e abra um terminal em cada uma. Na VM receptora (*receiver*), execute o script `run_receiver.sh` para iniciar o servidor `iPerf3` e coletar dados. Na VM remetente (*sender*), execute `run_sender.sh` para iniciar o cliente `iPerf3` e realizar as medições.

Após os testes de `iPerf3`, execute `run_formatter.sh` na VM receptora. Este script processa os dados brutos, gerando um arquivo `.csv` com os resultados finais no diretório de logs especificado.

6. Resultados

Os testes de desempenho de rede, realizados utilizando a ferramenta `iperf3`, revelaram uma diferença na vazão (throughput) entre os ambientes de Máquinas Virtuais (VMs) e contêineres Docker. Observou-se que a rede configurada com contêineres Docker alcançou uma vazão média de aproximadamente $51996.17 Mbps$, demonstrando uma performance substancialmente superior em comparação com a vazão média de $3740.15 Mbps$ obtida nas VMs. Esta notável diferença sublinha a eficiência inerente da arquitetura de contêineres devido ao seu menor overhead de virtualização e à utilização mais direta da pilha de rede do sistema operacional hospedeiro.

6.1. Geração de dados

A coleta de dados de desempenho de rede foi sistematicamente executada para ambos os ambientes, Máquinas Virtuais (VMs) e contêineres Docker, através de uma série de 30 ”rounds”de testes com a ferramenta `iperf3`. Para cada ambiente, um receptor `iperf3` foi configurado no modo `one-off` para gerar resultados em formato JSON. Este servidor registrava cada execução em arquivos JSON individuais, os quais eram subsequentemente concatenados em um único array JSON denominado `iperf3_combined.json`.

No lado do cliente, cada um dos 30 rounds de teste consistiu em uma execução de `iperf3` com duração de 10 segundos, configurada para exibir o throughput em megabytes. Um intervalo de 1 segundo foi introduzido entre cada round para garantir a estabilidade das medições. Esta abordagem round-robin permitiu a coleta de uma amostra robusta de dados de throughput para análise comparativa.

Finalmente, os dados brutos em formato JSON foram processados por um script de formatação que extraiu a vazão em bits por segundo e a converteu para megabytes por segundo. Estes valores, juntamente com o número do respectivo round, foram então organizados em um arquivo CSV, denominado `bps_values.csv`, servindo como a base para a análise quantitativa do desempenho de rede entre VMs e contêineres.

6.2. Análise de desempenho

A análise comparativa do desempenho de rede entre os ambientes de Máquinas Virtuais (VMs) e contêineres Docker revelou diferenças significativas na vazão (*throughput*) obtida. Conforme ilustrado no gráfico, que apresenta a distribuição da vazão para ambos os cenários, os contêineres Docker demonstraram uma capacidade de transferência de dados substancialmente superior. Enquanto as VMs apresentaram uma vazão média de aproximadamente $3740.15 Mbps$, as medições para os contêineres Docker atingiram uma média de cerca de $51996.17 Mbps$, indicando um desempenho de rede consideravelmente mais elevado. Esta disparidade de performance é um fator crítico para a compreensão das vantagens e desvantagens de cada tecnologia, e nos leva a uma reflexão aprofundada sobre as implicações práticas dessas arquiteturas para aplicações que demandam alta performance de rede.

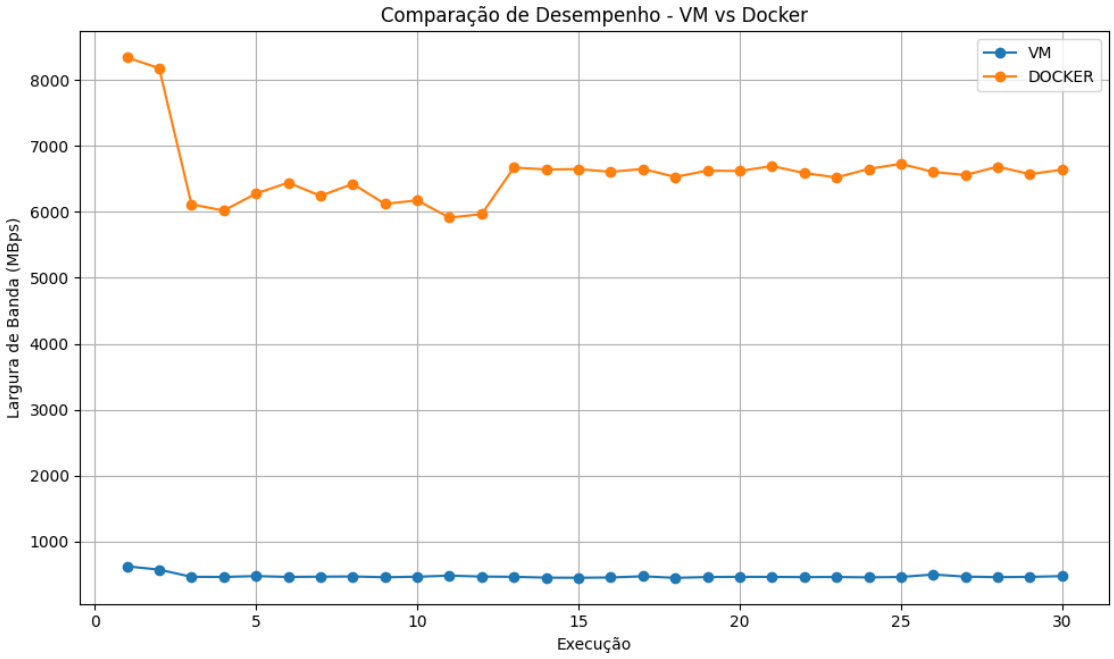

Figura 2. Comparação de desempenho - VM vs Docker

7. Conclusão

O experimento de análise de desempenho de rede, comparando VMs e contêineres Docker, demonstrou uma clara superioridade da tecnologia de contêineres em termos de vazão (throughput). As medições do `iperf3` revelaram que os contêineres Docker atingiram uma vazão média de $51996.17 Mbps$, enquanto as VMs obtiveram uma média de $3740.15 Mbps$. Esta diferença substancial, aliada a um coeficiente de variação percentual menor nos contêineres (7.79% versus 11.92% para VMs), indica não apenas uma

capacidade de transmissão de dados significativamente maior, mas também uma consistência relativa mais favorável, apesar das variações absolutas. A ausência de camadas de virtualização de hardware e a utilização direta do kernel do sistema operacional hospedeiro conferem aos contêineres um overhead consideravelmente menor, resultando em uma comunicação de rede mais eficiente.

Em suma, os resultados apontam que, para cenários onde a alta vazão de rede é um requisito primordial e o isolamento completo de hardware não é estritamente necessário, a implementação de aplicações em contêineres Docker oferece uma vantagem de desempenho notável em comparação com a virtualização tradicional. Embora as VMs ainda sejam indispensáveis para requisitos de isolamento de sistema operacional ou execução de múltiplas plataformas distintas, a eficiência de rede dos contêineres os posiciona como uma solução otimizada para arquiteturas modernas, como microsserviços e aplicações distribuídas, onde a performance e a agilidade são cruciais.

Referências

[Docker Inc. 2025a] Docker Inc. (2025a). Docker architecture. https://docs.docker.com/get-started/docker-overview/#docker-architecture.

[Docker Inc. 2025b] Docker Inc. (2025b). Requisitos do sistema operacional para instalar o docker engine no ubuntu. https://docs.docker.com/engine/install/ubuntu/#os-requirements.

[Felter et al. 2015] Felter, W., Ferreira, A., Rajamony, R., and Rubio, J. (2015). An updated performance comparison of virtual machines and linux containers. In *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 171–172. IEEE.

[Gomes et al. 2020] Gomes, L. D., de Mello, R. F., and de Rose, C. A. (2020). Performance evaluation of container-based virtualization for microservices architecture. In *2020 VIII Brazilian Symposium on Computing Systems Engineering (SBESC)*, pages 1–8. IEEE.

[Kozhirbayev and Sinnott 2017] Kozhirbayev, Z. and Sinnott, R. O. (2017). A performance evaluation of container-based virtualisation for high-performance computing environments. In *2017 25th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*, pages 388–395. IEEE.

[Leandro 2025] Leandro (2025). O que é e para que serve o VirtualBox? https://4infra.com.br/o-que-e-o-virtualbox/.

[Microsoft Azure 2025] Microsoft Azure (2025). What is a virtual machine and how does it work? https://azure.microsoft.com/en-us/resources/cloud-computing-dictionary/what-is-a-virtual-machine/.

[Muñoz 2019] Muñoz, S. (2019). The history of virtualization and its mark on data center management. https://www.techtarget.com/searchitoperations/feature/The-history-of-virtualization-and-its-mark-on-data-center-management.

[Oracle 2025] Oracle (2025). Oracle vm virtualbox. https://www.virtualbox.org/.

[Red Hat 2025] Red Hat (2025). What is a virtual machine (vm)? https://www.redhat.com/en/topics/virtualization/what-is-a-virtual-machine.

[Susnjara and Smalley 2024a] Susnjara, S. and Smalley, I. (2024a). O que é docker? https://www.ibm.com/br-pt/think/topics/docker.

[Susnjara and Smalley 2024b] Susnjara, S. and Smalley, I. (2024b). What is VMware? https://www.ibm.com/think/topics/vmware.

[VMware 2025] VMware (2025). What is a virtual machine? https://www.vmware.com/topics/glossary/content/virtual-machine.html.

Apêndices

A. Docker

```
name: docker_to_docker_comunication

services:
  receiver:
    container_name: receiver
    hostname: receiver
    networks:
      - default
    volumes:
      - receiver_volume:/local_bind
    environment:
      - DEBIAN_FRONTEND=noninteractive
      - TZ=UTC
    build: .
    command: ["bash", "-c", "
      echo 'Receiver initialized';
      ./run_receiver.sh;
      sleep infinity;
      "]

  sender:
    container_name: sender
    hostname: sender
    networks:
      - default
    volumes:
      - sender_volume:/local_bind
    environment:
      - DEBIAN_FRONTEND=noninteractive
      - TZ=UTC
      - RECEIVER_IP=receiver
    depends_on:
      - receiver
    build: .
    command: ["bash", "-c", "
      echo 'Sender initialized';
      ./run_sender.sh;
      ./run_formatter.sh;
      sleep infinity;
      "]

```

```
volumes:
  receiver_volume:
    driver: local
    driver_opts:
      o: bind
      type: none
      device: ./host_volume
  sender_volume:
    driver: local
    driver_opts:
      o: bind
      type: none
      device: ./host_volume

networks:
  default:
    driver: bridge
```

Listing 1. Configuração para docker compose usando modo de rede bridge.

```
name: docker_to_docker_comunication

services:
  receiver:
    container_name: receiver
    hostname: receiver
    network_mode: host
    volumes:
      - receiver_volume:/local_bind
    environment:
      - DEBIAN_FRONTEND=noninteractive
      - TZ=UTC
    build: .
    command: ["bash", "-c", "
      echo 'Receiver initialized';
      ./run_receiver.sh;
      sleep infinity;
      "]

  sender:
    container_name: sender
    hostname: sender
    network_mode: host
    volumes:
      - sender_volume:/local_bind
```

```
    environment:
      - DEBIAN_FRONTEND=noninteractive
      - TZ=UTC
      - RECEIVER_IP=localhost
    depends_on:
      - receiver
    build: .
    command: ["bash", "-c", "
      echo 'Sender initialized';
      ./run_sender.sh;
      ./run_formatter.sh;
      sleep infinity;
      "]

volumes:
  receiver_volume:
    driver: local
    driver_opts:
      o: bind
      type: none
      device: ./host_volume
  sender_volume:
    driver: local
    driver_opts:
      o: bind
      type: none
      device: ./host_volume
```

Listing 2. Configuração para docker compose usando modo de rede host.

```
FROM debian:12.11

WORKDIR /local_bind

RUN ln -fs /usr/share/zoneinfo/UTC /etc/localtime && echo "UTC" >
 ↪  /etc/timezone
RUN apt update && apt install -y \
    iperf3 \
    net-tools \
    iputils-ping \
    jq
RUN apt clean && rm -rf /var/lib/apt/lists/*
```

Listing 3. Configuração do Dockerfile.

```bash
#!/bin/bash

hostname -I

LOG_DIR="/local_bind/iperf3_receiver_logs"

rm -rf "$LOG_DIR"
mkdir -p "$LOG_DIR"

ARRAY_FILE="$LOG_DIR/iperf3_combined.json"
echo "[" > "$ARRAY_FILE"

FIRST=1

while true; do
    TIMESTAMP=$(date +"%Y-%m-%d_%H-%M-%S")
    LOG_FILE="$LOG_DIR/iperf3_${TIMESTAMP}.json"
    iperf3 -s --one-off  --json 2>&1 | tee "$LOG_FILE"
    if [ $FIRST -eq 1 ]; then
        cat "$LOG_FILE" >> "$ARRAY_FILE"
        FIRST=0
    else
        echo "," >> "$ARRAY_FILE"
        cat "$LOG_FILE" >> "$ARRAY_FILE"
    fi
done
```

Listing 4. Código para o contêiner que atuará como servidor recebendo dados.

```bash
#!/bin/bash

for i in {1..30}; do
    iperf3 -c "$RECEIVER_IP" -t 10 --format M --verbose
    echo "Run $i completed"
    sleep 1
done

ARRAY_FILE="/local_bind/iperf3_receiver_logs/iperf3_combined.json"
echo "]" >> "$ARRAY_FILE"

echo "Finalized all runs. Results saved to $ARRAY_FILE"
```

Listing 5. Código para o contêiner que atuará como cliente enviando dados.

```
#!/bin/bash

echo "Started formatting iperf3 results..."

ARRAY_FILE="/local_bind/iperf3_receiver_logs/iperf3_combined.json"

OUTPUT_FILE="/local_bind/iperf3_receiver_logs/bps_values.csv"

echo "run_number,megabytes_per_second" > "$OUTPUT_FILE"
jq -r 'to_entries[] |
 ↪  "\(.key|tonumber+1),\(.value.end.sum_received.bits_per_second / 8
 ↪  / 1000000)"' "$ARRAY_FILE" >> "$OUTPUT_FILE"

echo "Formatted results saved to $OUTPUT_FILE"
```

Listing 6. Código para formatação dos dados gerados.

```
import pandas as pd
import matplotlib.pyplot as plt

# Lê o CSV
df = pd.read_csv("iperf3_resultados.csv")

# Converte o timestamp para uma data legível
df["timestamp"] = pd.to_datetime(df["timestamp"], unit="s")

# Gráfico de comparacao

plt.figure(figsize=(10, 6))
for env in df["env"].unique():
dados = df[df["env"] == env]
plt.plot(dados["timestamp"], dados["bandwidth_Mbps"], marker="o",
 ↪  label=env.upper())

plt.xlabel("Horário do Teste")
plt.ylabel("Largura de Banda (Mbps)")
plt.title("Comparação de Desempenho - VM vs Docker")
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.savefig("grafico_comparativo.png")
plt.show()
```

Listing 7. Código usado para a geração de gráficos.