Avaliação de Soluções de Emulação Leve e Distribuída para Experimentação de Rede

Emerson R. A. Barea[1], **César A. C. Marcondes**[2], **Hermes Senger**[3] **e Diego F. Pedroso**[3]

[1]Departamento de Informática (DInf) - Instituto Federal do Tocantins (IFTO)
[2]Divisão de Ciência da Computação (IEC) - Instituto Tecnológico de Aeronáutica (ITA)
[3]Departamento de Computação (DC) - Universidade Federal de São Carlos (UFSCar)

emerson.barea@ifto.edu.br, cmarcondes@ita.br e

{hermes.senger,diego.pedroso}@ufscar.br

Resumo. *Apesar das soluções de emulação leve e distribuída utilizadas na experimentação de rede normalmente seguirem o conceito de virtualização por container, diferenças de implementação podem gerar comportamento distinto quanto ao consumo de recursos computacionais e escalabilidade suportada. Nessa linha, este trabalho avalia experimentalmente o Mininet Cluster e Maxinet com a emulação de topologias de datacenter com requisitos distintos quanto ao número de elementos criados e o tamanho do cluster. Os resultados apontam diferenças relevantes no consumo de memória, número de processos criados e conexões estabelecidas entre os nós do cluster, apresentando características que auxiliam identificar a tecnologia mais indicada para cenários específicos.*

Abstract. *Although typically the lightweight and distributed emulation solutions used in network experimentation follow the container virtualization concept, implementation differences can generate distinct behavior regarding computational resources consumption and supported scalability. Thus, in this work we experimentally evaluates the Mininet Cluster and Maxinet with the emulation of data center topologies with different requirements regarding the number of elements created and the size of the cluster. Results point to relevant differences in memory consumption, number of processes created and connections established between cluster nodes, showing characteristics that help identify the technology best suited for specific scenarios.*

1. Introdução

Os métodos de experimentação para estudo das tecnologias de rede geralmente envolvem algum tipo de emulação [Yan and McKeown 2017], pois a emulação de rede atende as vantagens oferecidas pelas redes físicas, suportando o uso de aplicações reais, sem o ônus do alto custo e da complexidade do ambiente físico, além de garantir maior precisão nos resultados quando comparada à simulação [White et al. 2002] [Huang et al. 2014].

Com o passar do tempo, novas técnicas de emulação foram empregadas. Atualmente, as soluções comumente utilizadas suportam a emulação completa de ambientes de rede complexos usando virtualização leve baseada em *containers* hospedados sobre uma única máquina. No entanto, embora esse tipo de emulação seja amplamente utilizada e

seus benefícios sejam bem conhecidos, há cenários em que os requisitos computacionais exigidos pela rede emulada são superiores aos suportados pela máquina individual, seja em número de nós emulados, taxa de transferência máxima desejada ou outro parâmetro importante ao experimentador [Yan and McKeown 2017].

Essa limitação é considerada crítica e há tempos vem sendo estudada. Algumas soluções mistas que envolvem emulação com simulação foram propostas para resolver o problema da escalabilidade [Liu et al. 2015] [Yan and Jin 2015], no entanto, as soluções baseadas somente na emulação em *containers* também evoluíram e agora suportam o processamento distribuído. Embora existam, essas soluções ainda não tiveram suas características de escalabilidade devidamente avaliadas, dificultando a definição de qual tecnologia atende cenários específicos ou quais recursos computacionais são necessários para a experimentação da rede.

Baseados no cenário exposto, nesse trabalho nos concentramos em avaliar o comportamento de algumas dessas soluções de emulação baseadas em *containers*, que suportam processamento distribuído e podem ser usadas na experimentação de rede, quanto a utilização de recursos computacionais básicos, como memória, quantidade de processos criados e número de conexões de rede estabelecidas entre as máquinas que compõem o ambiente computacional distribuído. Entre os destaques e aspectos mais relevantes, nesse trabalho *(i)* analisamos preliminarmente algumas características técnicas de implementação dessas soluções para identificar qual tecnologia proporciona menor consumo de recursos computacionais gerais; e *(ii)* avaliamos o comportamento das ferramentas Mininet Cluster e MaxiNet em relação à utilização de recursos computacionais em cenários com requisitos específicos, identificando benefícios e desvantagens de cada implementação.

O desenvolvimento desse trabalho justifica-se pela importância de comparar aspectos técnicos de soluções de virtualização leve e distribuída para experimentação em rede, gerando dados que auxiliem na identificação de qual solução é indicada para cada demanda, facilitando o reconhecimento de requisitos exigidos por diferentes cenários. Da mesma forma, esse trabalho não tem como foco avaliar a acurácia dos resultados dessas soluções quando comparados aos resultados obtidos em ambientes reais, pois, consideramos essa fase já atendida pela maturidade dessas ferramentas e diversas adequações propostas em trabalhos anteriores [Beshay et al. 2015] [Cao et al. 2017] e [Ortiz et al. 2016].

O restante deste trabalho está organizado da seguinte forma: a seção 2 apresenta trabalhos relacionados e conceitos preliminares que servirão como base de informação para o desenvolvimento do trabalho. Na seção 3 são apresentados os requisitos a serem atendidos pela experimentação de rede. Na seção 4 há o detalhamento do procedimento e tecnologias utilizadas na avaliação das soluções de emulação leve e distribuída. Na seção 5 são apresentados e comentados os resultados dos testes realizados; e a seção 6 conclui e apresenta sugestões de trabalhos futuros.

2. Fundamentação Teórica e Trabalhos Relacionados

Esta seção apresenta conceitos e definições preliminares baseadas em trabalhos relacionados que são consideradas no desenvolvimento deste trabalho. Inicialmente é apresentada a linha de evolução da virtualização com atenção às características relevantes à experimentação em rede (subseção 2.1), finalizando com o detalhamento técnico de algumas soluções baseadas em virtualização leve e distribuída (subseção 2.2).

2.1. Virtualização Leve

O conceito de virtualização não é algo recente e tanto as plataformas quanto a camada de software responsável por abstrair e compartilhar o hardware com os níveis de isolamento necessário entre as funções virtualizadas evoluíram, passando da necessidade de ter S.Os. completos virtualizados sobre hipervisores com alta sobrecarga de funções, para a virtualização leve baseada no isolamento de processos na área de usuário sobre um mesmo kernel (*containers*) [Ganesh et al. 2012], fazendo uso do conceito de *namespaces* no isolamento de funções e *Control Groups* (CGroups) no controle de recursos [Daniels 2009].

De maneira geral, a finalidade de cada *namespace* é envolver um recurso de sistema global em uma abstração que faça parecer aos processos que eles têm sua própria instância isolada desse recurso, dando a ilusão de que são os únicos no sistema. Para isso, cada *namespace* faz uso da atribuição de um valor ao parâmetro CLONE_NEW(), onde () corresponde ao identificador exclusivo do *namespace*, que é passado às três chamadas de sistema (*clone()*, *unshare()* e *setns()*) responsáveis pela criação ou ligação de cada processo ao *namespace* correspondente [Kerrisk 2013]. Já o CGroup é uma solução de gerenciamento de recursos que fornece um mecanismo para agregar ou particionar conjuntos de processos, e processos filhos, em grupos hierárquicos, também denominados subsistemas ou controladores, com comportamento especializado. Este comportamento está relacionado à limitação, priorização, contabilização e controle da alocação dos recursos da máquina, como memória, CPU, dispositivos e I/O, aos respectivos grupos [Brown 2014].

Devido suas características, a virtualização por *container* mostrou-se viável como base para a experimentação em rede, principalmente por suportar a emulação de ambientes inteiros que possibilitam o estudo de cenários variados e complexos, com custo geral bastante reduzido e precisão nos resultados [Yan and McKeown 2017]. Atualmente existe grande quantidade de ferramentas que implementam esse tipo de virtualização, e algumas delas são bem conhecidas e amplamente utilizadas nos mais variados ambientes.

O LXC é um exemplo clássico de virtualização leve baseada em *container*. Faz uso dos *namespaces* de IPC, UTS, PID, rede e usuários, garantindo controle dos recursos do *container* mesmo quando criado em modo não-privilegiado. Implementa *chroot* criando uma abstração do ponto de montagem do sistema de arquivos raiz do *container* (*pivot_root*), utiliza CGroups e outros mecanismos de segurança e isolamento do processo, possui uma API responsável pela integração com a *liblxc* no gerenciamento de todo ciclo de vida e controle dos *containers*, além de suportar a criação de novas instâncias a partir de imagens de S.O. diferentes do utilizado na máquina hospedeira, desde que suportado pelo kernel e respeitada a arquitetura [Canonical a].

Outra solução baseada no conceito de virtualização leve é o *LinuX container Daemon* (LXD), que consiste num serviço que expõe uma API REST através de um *socket* Unix ou acessível via rede, que faz uso de uma ferramenta de linha de comando para gerenciar *containers* LXC via *liblxc*. O LXD é basicamente uma alternativa às ferramentas de administração LXC, com algumas funções extras, como acesso direto a dispositivos da máquina hospedeira, tranferência de imagens e *containers* entre máquinas distintas, gerenciamento de rede e armazenamento. Além da API padrão, o LXD ainda é suportado pelo projeto *nova-lxd* do OpenStack [Canonical b].

Assim como o LXD, o Docker também surgiu como um serviço acessível via API

REST através de *socket* Unix e rede para administração de *containers* LXC. Passou a utilizar a *libcontainer* no gerenciamento direto de *containers* a partir da versão 0.9, tornando opcional o uso de *libvirt*, LXC ou *systemd-nspawn* [Hykes 2014]. Suporta o download de imagens pré-configuradas de seu repositório (*Docker Hub*) e o empacotamento de *containers* com suas dependências para criação de novas imagens. Suporta *copy-on-write* com o compartilhamento do sistema de arquivos entre *containers* (UnionFS), onde somente o conteúdo modificado é replicado entre as diferentes instâncias, aumentando seu desempenho enquanto diminui o consumo de disco da máquina hospedeira [Merkel 2014].

Além das soluções baseadas em *containers* que implementam um grande conjunto de mecanismos de isolamento, existem também ferramentas desenvolvidas com foco específico na emulação para experimentação em rede, como o Mininet. O Mininet diferencia-se de outras implementações principalmente por utilizar apenas os *namespaces* de rede e de ponto de montagem atrelados a um processo Linux, conferindo menor sobrecarga de funções em cada *container*. Adicionalmente, faz uso de CGroups para controle de recursos, suporta o gerenciamento de switches e controladores de SDN com uso do *Open vSwitch*, interconectando todos elementos da rede emulada com interfaces virtuais (*veth*). Possui uma API acessível por linha de comando ou programável, que possibilita a administração de todo ciclo de vida e controle dos elementos de um experimento.

As características apresentadas propiciam a essas soluções baixo consumo de recursos computacionais, porém, como visto, mesmo compartilhando o conceito de *containers*, possuem diferenças de implementação que acarretam diferença de comportamento quanto ao consumo de recursos computacionais, como apresentado na Figura 1. A Figura 1 apresenta a relação de consumo de memória, Figura 1(a), número de processos Linux, Figura 1(b), e consumo de espaço em disco, Figura 1(c), em cenários com a criação de apenas 1 e 100 *containers* com as tecnologias Mininet, LXC, LXD e Docker.

Como o LXC, LXD e Docker utilizam imagens de S.O. na criação dos *containers*, optou-se por utilizar as menores e mais simples imagens disponíveis nos repositórios oficiais de cada ferramenta, consumindo o mínimo possível de recursos na criação de *containers* básicos com cada tecnologia. Para isso, foram utilizadas as imagem do Linux Alpine 3.4 AMD64[1] com LXD e Linux BusyBox 1.29 AMD64[2] com LXC e Docker.

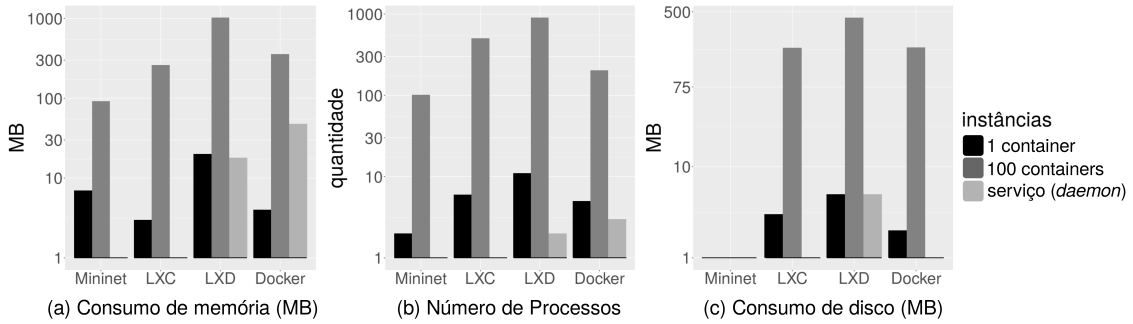

Figure 1. Consumo de recursos computacionais pelas soluções de virtualização leve.

[1]https://alpinelinux.org/
[2]https://busybox.net/

Analisando os gráficos de consumo de recursos da Figura 1 é possível observar que o Mininet consome aproximadamente 90 MB de memória RAM da máquina hospedeira quando instanciados 100 *containers*, enquanto o LXC consome 260MB, o Docker 350MB e o LXD 1.02GB. Este consumo está diretamente relacionado às características das imagens de S.O. utilizadas na criação dos *containers*, pois, ela carrega consigo todo conjunto de códigos, bibliotecas, variáveis de ambiente e arquivos de configuração necessários para seu funcionamento. Quanto a quantidade de processos criados para instanciação de 100 *containers*, o Mininet cria apenas 101 processos, o Docker 202, o LXC 501 e o LXD 902 processos.

Quanto ao consumo de disco, o Mininet não faz uso algum, visto que seus *containers* utilizam um mesmo ponto de montagem comum a todos usuários no sistema de arquivos da máquina hospedeira, enquanto o LXC e LXD consomem respectivamente 210 MB e 430 MB cada, devido à criação de sistemas de arquivos virtuais independentes para cada *container*. Já o Docker, consome aproximadamente 140MB de espaço em disco para criação dos 100 *containers*, devido a utilização da técnica de *copy-on-write*.

Com base nos dados apresentados, é possível observar que o Mininet consome menos recursos gerais da máquina hospedeira no processo de criação de *containers* simples quando comparado às outras tecnologias apresentadas.

2.2. Virtualização Leve e Distribuída

Quando o conceito de virtualização leve é expandido para ambiente distribuído, é necessário analisar também os mecanismos para conexão entre *containers* em máquinas distintas e as técnicas suportadas para ajuste dos parâmetros de banda e atraso dos links.

O LXD, por exemplo, suporta a criação de túneis GRE e VXLAN para comunicação entre *containers* em máquinas distintas, porém, todo processo de criação dos túneis e distribuição dos *containers* não é automatizado, havendo necessidade de ser definido e solicitado pelo utilizador através da API. Outra forma é a integração com o projeto *Neutron* do OpenStack, porém, esse procedimento representa aumento significativo de complexidade e consumo de recursos computacionais do ambiente. A API do LXD não suporta a configuração dos parâmetros de banda e atraso de links [Graber 2016].

O Docker possui um ecossistema para gerenciamento de servidores em *cluster*, *Docker Cloud*, que suporta automatização da estratégia de distribuição de *containers* em seus nós através dos algoritmos *Emptiest node*, que distribui os *containers* nos nós menos utilizados no momento da criação; *High availability*, cria os *containers* de um mesmo serviço num único nó considerado menos utilizado no momento da criação; e o *Every node*, que cria um *container* em cada nó participante do *cluster*. A interconexão entre *containers* em máquinas distintas é feita utilizando *plugins*, que fazem uso da *libnetwork* e suportam a configuração de parâmetros de rede de acordo com a necessidade do utilizador; ou redes *overlay*, através de túneis VXLAN gerenciados pelo *Swarm*[3]. O *Swarm* não suporta parametrização de banda e atraso de links nativamente [Docker 2018].

Quanto ao Mininet, passou a suportar processamento distribuído utilizando túneis SSH na comunicação entre *containers* em máquinas distintas na versão 2.2.0, e túneis GRE, que suportam maior vazão por não depender do TCP no transporte de dados, na

[3]https://docs.docker.com/engine/swarm/

versão 2.3.0. A criação dos túneis nas máquinas hospedeiras é feita automaticamente quando informada a existência de um link entre dois elementos Mininet em sua API, porém, esse procedimento não suporta a configuração dos parâmetros de banda e atraso dos links [Lantz and O'Connor 2015]. Possui algoritmos pré-definidos para distribuição automática dos elementos nos nós do *cluster*, sendo que, o *SwitchBinPlacer* distribui switches e controladores em blocos de tamanho uniforme com base no tamanho do *cluster*, tentando alocar os *containers* de hosts no mesmo servidor em que estão os switches; e o *RandomPlacer* faz a distribuição randômica dos elementos pelo *cluster* [Burkard 2014].

Outra implementação de Mininet distribuído é o MaxiNet, que consiste em uma API atuando como uma camada sobre o Mininet, onde um nó central, denominado *Frontend*, invoca comandos nos nós remotos, *workers*, gerenciando os elementos Mininet localizados na rede por um servidor de nomes de objetos remotos do Python, o *PYthon Remote Object* versão 4 (Pyro4). Utiliza túneis GRE para conexão remota entre elementos Mininet, porém, esses túneis são suportados apenas por elementos do tipo *switch*. Suporta a configuração dos parâmetros de banda e atraso dos links emulados diretamente em sua API e cria automaticamente os túneis GRE nas máquinas hospedeiras. Faz uso da biblioteca METIS[4] para particionamento do grafo da topologia emulada, criando partições com pesos equivalentes em todos nós do *cluster*, confinando a maior parte do tráfego emulado localmente nos *workers* através do critério de corte mínimo baseado na banda dos links da topologia [Wette et al. 2014].

3. Requisitos para Experimentação de Rede

De maneira geral, as soluções apresentadas na seção 2 possuem características relevantes que as viabilizam como tecnologia base para experimentação de rede, porém, algumas delas possuem em sua estrutura recursos acima do necessário para essa finalidade. Além disso, essas funções podem gerar algum tipo de sobrecarga que resultará em perda de desempenho ou até mesmo consumo desnecessário dos recursos computacionais do ambiente de experimentação.

Com base no exposto, este trabalho propõe os seguintes requisitos para soluções de emulação leve e distribuída para experimentação de rede:

- *Requisitos básicos*. Estão relacionados ao suporte à virtualização leve e distribuída baseada em *containers*, suportando processamento em várias máquinas da rede. O isolamento das funções deve garantir o processamento necessário, bem como impedir acessos indevidos aos dados relacionados a cada função. Deve suportar emulação de aplicações e protocolos reais de rede.
- *Sobrecarga de funções*. A solução deve implementar o mínimo necessário de funções, garantindo somente os recursos estritamente necessários à experimentação de rede, sendo desnecessário todo e qualquer função extra, como implementação de *namespaces* não relacionados aos serviços de rede, por mais simples que seja. Esse requisito visa garantir o mínimo consumo de recursos computacionais do ambiente de experimentação.
- *Parametrização*: A solução deve suportar a parametrização do ambiente, de forma a garantir que as redes emuladas representem fielmente as necessidades do experimentador, possibilitando a experimentação de cenários variados utilizando um

[4]http://glaros.dtc.umn.edu/gkhome/metis/metis/overview

mesmo ambiente físico. Além disso, a solução deve possuir mecanismos que suportem a distribuição dos elementos da topologia no ambiente computacional, utilizando algoritmos ou processos automatizados e configuráveis.

Ao confrontar as tecnologias de virtualização leve apresentadas na seção 2 com os requisitos aqui definidos, identificou-se que o Mininet, em suas duas distribuições que suportam processamento distribuído, é a solução que atende aos requisitos de maneira mais completa. Sendo assim, tanto o Mininet Cluster quanto o MaxiNet são as soluções de emulação leve e distribuída para experimentação de rede identificadas para avaliação.

4. Materiais e Métodos

Nesta seção são detalhados o método utilizado na avaliação das soluções de emulação leve e distribuída de rede, apresentando o procedimento utilizado e parâmetros ajustados em cada tecnologia (subseção 4.1), tipos de topologias emuladas e estratégia para particionamento em ambiente distribuído (subseção 4.2), técnica de planejamento de experimento e análise dos resultados (subseção 4.3) e tecnologias utilizadas (subseção 4.4).

4.1. Procedimento e Parâmetros de Medição

A avaliação foi baseada na análise do comportamento e consumo de recursos em um *cluster* com as tecnologias Mininet Cluster e MaxiNet. Para isso, foram emuladas redes de *datacenter* com topologias compostas pelos elementos *host* do Mininet, representando os servidores da rede; *switch*, correspondendo aos switches utilizados na interligação dos servidores; e *link*, responsáveis pela conexão entre servidores e switches. Nos cenários testados houve variação da quantidade de elementos Mininet criados, nós do *cluster* alocados e tecnologia Mininet utilizada para processamento distribuído.

Durante todo processo de criação e ativação dos elementos da topologia, monitorou-se o efeito percebido pela variação de um único fator, ou um conjunto deles, na utilização de memória RAM, quantidade de processos Linux gerados e conexões de rede estabelecidas entre os nós físicos do *cluster*.

4.2. Topologias e Particionamento

Foram emuladas duas topologias de *datacenter* com diferentes requisitos computacionais. A primeira é a FatTree, Figura 2(a), que segue a estrutura de uma árvore k-nária, sendo reconhecida por suportar alta capacidade de banda passante entre os nós da rede utilizando switches comuns. É composta por k grupos de switches (*pods*), cada um contendo duas camadas com $k/2$ switches cada (*Aggregate* e *Edge*), com k portas por switch, e $(k/2)^2$ servidores. Cada *Edge* switch está conectado a $k/2$ servidores e $k/2$ *Aggregate* switches. Acima dos *pods* há uma outra camada (*Core*) com $(k/2)^2$ switches, cada um conectado aos k-*pods* via *Aggregate* switches [Al-Fares et al. 2008]. Como exemplo, uma topologia FatTree $k = 4$ possui 4 *pods* compostos por 4 servidores, 2 *Edge* switches e 2 *Aggregate* switches cada, mais 4 *Core* switches, totalizando 16 servidores, 20 switches e 144 links em toda topologia.

Já a topologia DCell, Figura 2(b) é formada por k células (*cells*) compostas por t servidores cada, onde $t = k - 1$. Os t servidores de uma *cell* estão interconectados através de switches independentes à cada *cell*, bem como cada servidor$_{k|t}$ está conectado diretamente a outro servidor$_{t+1|k}$ [Guo et al. 2008]. Como exemplo, uma topologia DCell

$k = 4$ possui 4 *cells* compostas por 3 servidores e 1 switch cada, totalizando 4 switches, 12 servidores e 18 links em toda topologia.

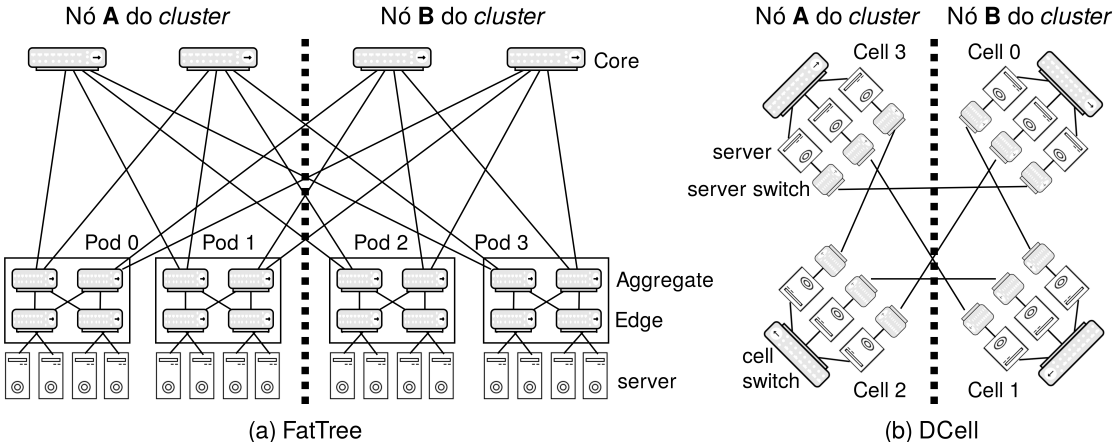

Figure 2. Representação das topologias FatTree (a) e DCell (b) utilizadas na avaliação das soluções de Mininet distribuído.

Buscando garantir a distribuição igualitária da carga de processamento no *cluster*, foi utilizada a técnica de *Round Robin* no particionamento dos k elementos da topologia entre seus nós, alocando também todos elementos de um único *pod*, na topologia FatTree, e mesma *cell*, na topologia DCell, num mesmo servidor.

Devido ao fato do MaxiNet suportar comunicação remota apenas entre elementos Mininet do tipo *switch*, como apresentado na subseção 2.2, foram incluídos switches nos links entre servidores nessa topologia, identificados como $serverswitch$ na Figura 2(b).

4.3. Metodologia de Testes

A avaliação experimental das soluções de Mininet distribuído depende do ajuste dos parâmetros de fatores de controle, que são testados simultaneamente e podem assumir diferentes níveis. Para cada combinação dos possíveis níveis, é necessário executar uma ou mais sequências de testes que geram resultados para serem analisados, porém, a definição do ajuste em cada parâmetro não é algo trivial, bem como o aumento na quantidade de fatores e níveis testados aumenta consideravelmente o tamanho do experimento, podendo até inviabilizar a obtenção de resultados significativos.

Uma alternativa para experimentação em ambientes com essa característica é a utilização da técnica de planejamento fatorial [Montgomery 2006]. Planejamento fatorial possibilita medir os efeitos, ou influências, de uma ou mais variáveis na resposta de um processo. Essa técnica consiste na identificação de um conjunto finito de fatores que influenciam o comportamento do ambiente, atribuindo valores específicos e válidos aos níveis de cada fator, o que altera o resultado das variáveis monitoradas. A relação entre os fatores e níveis é do tipo exponencial, $(niveis)^{fatores}$, e sua ocorrência mais comum é a 2^k fatorial, onde um conjunto de k fatores assume dois níveis de valores possíveis cada.

Partindo deste princípio, para análise experimental das soluções de Mininet distribuído foram identificados uma sequência de 3 fatores de controle com 2 níveis de variação cada, formando a representação de um experimento do tipo 2^3 fatorial. Figura 3(a). O primeiro fator corresponde ao tipo de tecnologia Mininet distribuído

($MininetDistribuido$), com níveis variando entre MaxiNet (MN) e Mininet Cluster com links GRE (MC). O segundo fator é o tamanho da topologia ($TamanhoTopologia$), que corresponde à quantidade de *pods*, na topologia FatTree, e *cells*, na topologia DCell, com níveis 4 e 12. Tal variação visa garantir a análise experimental em topologias compostas por poucos e muitos elementos. O terceiro fator é a quantidade de nós do *cluster* ($Tamanhocluster$), com níveis 2 e 4, dessa forma, todo ambiente é testado em um *cluster* composto por 2 e 4 nós. Cada fator recebe uma denominação do tipo $x1$, $x2$, ..., xn, que simplifica sua identificação.

		Níveis			
	Fatores	-1	+1	Resultados	
x1	Mininet Distribuído	MN	MC	RAM	proc.
x2	Tamanho topologia	4	12	RAM	proc.
x3	Tamanho *cluster*	2	4	RAM	proc.

(a) Tabela de Fatores

	Fatores de Controle			
Teste	x1	x2	x3	Seq.
1	-1	-1	-1	5
2	+1	-1	-1	7
3	-1	+1	-1	1
4	+1	+1	-1	8
5	-1	-1	+1	2
6	+1	-1	+1	3
7	-1	+1	+1	4
8	+1	+1	+1	6

(b) Matriz de Planejamento

Figure 3. Tabela de Fatores (a) e Matriz de Planejamento (b) para experimentação das soluções de Mininet distribuído utilizando 2^3 fatorial.

No planejamento de um experimento fatorial 2^k, os dois níveis de cada fator são denominados nível baixo e nível alto, podendo ser identificados com os valores (-1) no nível baixo e (+1) no nível alto, conforme apresentado na coluna de níveis da Figura 3(a). A partir dessa definição foi possível identificar as combinações de testes do experimento, Figura 3(b), cuja definição da distribuição dos níveis dos fatores no plano seguiu o procedimento:

- Para $x1$, o sinal da coluna de (1) alterna em grupos de $2^0 = 1$, ou seja, seguidamente.
- Para $x2$, o sinal da coluna de (1) alterna em grupos de $2^1 = 2$, ou seja, em pares.
- Para $x3$, o sinal da coluna de (1) alterna em grupos de $2^2 = 4$, ou seja, 4 vezes (-1) seguidos de 4 vezes (+1).

Por fim, foi definido a sequência aleatória para execução de cada teste do experimento, minimizando a possibilidade de interferência de fontes de variação não assinaláveis no ambiente. Concluídas essas etapas, o experimento pôde então ser executado.

4.4. Ambiente e Tecnologias

O *cluster* utilizado no experimento é composto por 4 servidores. Cada um possui 1 processador com 8 núcleos de 2.40GHz, 8GB de memória RAM e duas interfaces de rede de 1Gbps cada. O S.O. é o Ubuntu Server 16.04.5 LTS, kernel 4.4.0-138, e os principais programas instalados são Mininet 2.3.0d4, MaxiNet 1.2, *Open vSwitch* 2.5.5 e Pyro4.

Cada servidor estava conectado em uma rede de experimentação totalmente isolada, composta por um switch com interfaces de 1Gbps em topologia estrela. Essa rede é utilizada exclusivamente para troca de mensagens das soluções de Mininet distribuído na gestão do ciclo de vida do experimento. A segunda interface de rede de cada servidor

estava conectada em um segundo switch com interfaces de 1Gbps, também em topologia estrela, numa rede independente para administração dos servidores. A topologia física utilizada na interligação dos nós do *cluster* buscou impedir a interferência de fatores externos não previstos nos resultados do experimento.

5. Testes de Validação e Discussão dos Resultados

Após as definições apresentadas na seção 4, foram realizados os testes para avaliação das soluções de Mininet distribuído seguindo a sequência definida durante o planejamento do experimento, Figura 3(b), com um total de 10 replicações para cada cenário.

A Figura 4 apresenta os resultados das medições realizadas, apontando a utilização total de memória RAM do ambiente, número de processos Linux criados e conexões de rede estabelecidas entre os nós do *cluster* durante todo processo de criação dos elementos Mininet e recursos adicionais necessários em cada cenário avaliado.

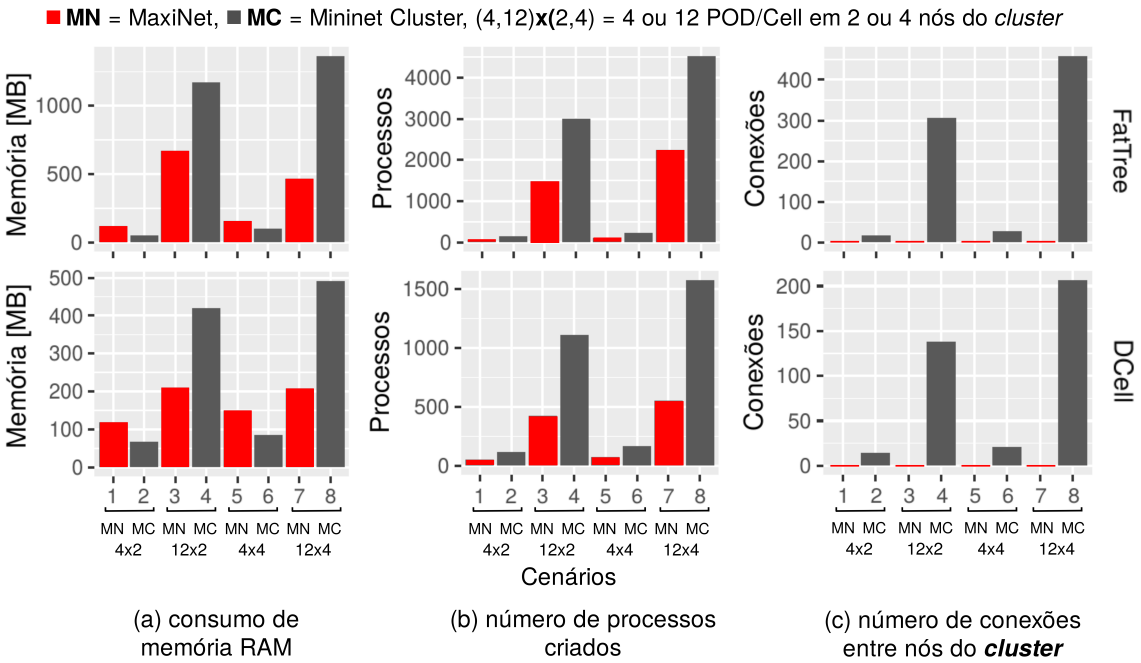

Figure 4. Consumo de memória, número de processos Linux criados e conexões de rede estabelecidas entre os nós do *cluster* durante a experimentação para topologias FatTree e DCell.

De forma geral, é possível observar que o total de memória utilizada no *cluster* variou consideravelmente entre todos cenários contendo 4 e 12 *pods* ou *cells*, Figura 4(a), partindo de aproximadamente 55MB de memória RAM utilizada no cenário 2 da topologia FatTree, cenário composto por 4 *pods* distribuídos em 2 nós de *cluster* utilizando Mininet Cluster, para até 1.4GB de memória utilizada no cenário 8 da mesma topologia, que conta com 12 *pods* distribuídos em 4 nós de *cluster* utilizando Mininet Cluster.

Por outro lado, também é possível observar variação dos resultados, mesmo em menor escala, entre cenários contendo o mesmo número de nós de *cluster*, *pods* ou *cells*, variando apenas a tecnologia de Mininet distribuído. Um exemplo desse comportamento pode ser visto entre os cenários 7 e 8 da topologia DCell. O cenário baseado em MaxiNet

apresenta consumo de memória RAM equivalente a somente 42% quando comparado ao Mininet Cluster. Tal fato ocorre devido à forma como as duas tecnologias administram os elementos Mininet remotos. O Maxinet, como apresentado na subseção 2.2, utiliza o Pyro4 para gerenciar os objetos Python remotos, portanto, novos elementos Mininet são solicitados diretamente ao servidor Pyro4 que se encarrega de enviar a solicitação ao Mininet da máquina remota para execução. Já o Mininet Cluster cria uma nova conexão SSH independente entre o nó *master*, máquina onde o Mininet está sendo executado inialmente, e o nó *slave*, máquina onde o elemento deve ser criado, carregando o código $mnexec$ responsável pela criação do elemento remoto. Esse túnel SSH permanece estabelecido enquando o elemento remoto existir. Tal procedimento consome memória, processos e conexões entre os nós do *cluster* envolvidos no processo.

O fato do MaxiNet utilizar o Pyro4 no gerenciamento de objetos Python, gera a necessidade de que esse serviço esteja em execução permanente em todos nós do *cluster*, antes mesmo que qualquer elemento Mininet local ou remoto seja criado. Isso justifica o consumo superior de memória RAM do MaxiNet em cenários contendo poucos elementos Mininet, como por exemplo, entre os cenários 1 e 2 das topologias FatTree e DCell. Nesse caso, o serviço Pyro4 consome aproximadamente 55MB de memória RAM, no nó *FrontEnd*, e 25MB de memória RAM em cada nó *worker*, enquanto o Mininet Cluster não depende ou necessita de serviço prévio.

Quanto aos processos Linux criados em cada cenário, Figura 4(b), sua variação também tem relação direta com o número de elementos Mininet da topologia, pois, cada elemento local ou remoto faz uso de pelo menos 1 novo processo Linux. Já o número de conexões entre os nós do *cluster*, Figura 4(c), tem relação direta com a tecnologia Mininet utilizada. Nesse caso, como o Mininet Cluster cria túneis SSH independentes entre os nós *master* e *slave* a cada novo elemento remoto, esse comportamento reflete no grande número de processos Linux e conexões em topologias maiores. O MaxiNet, por sua vez, cria um número de processos Linux diretamente proporcional ao número de elementos da topologia, além de apresentar poucas conexões remotas devido ao fato delas serem responsáveis apenas pela troca de mensagens entre objetos do Pyro4. Esse comportamento explica a diferença relevante do número de processos e conexões em cenários com o mesmo número de elementos e nós de *cluster*, variando apenas a tecnologia Mininet utilizada, como pode ser observado, por exemplo, nos cenários 7 e 8 das topologias FatTree e DCell.

Outra análise importante é o efeito causado pela variação isolada de cada fator, ou um conjunto deles, no resultado. Essa análise ajuda identificar a melhor estratégia para escalar as topologias de rede emuladas, consumindo o mínimo de recursos do *cluster*. A Figura 5(a) apresenta o efeito de cada fator independe, também conhecido como efeito principal. Esse efeito é analisado observando a variação do consumo de memória, que corresponde ao grau de inclinação da reta que representa o fator, quando passado do nível (-1) para o nível (+1). Como resultado, é possível notar no gráfico apresentado na Figura 5(a), que o fator com maior efeito é o $TamanhoTopologia$, seguido pelo $MininetDistribuido$, bem como o $Tamanhocluster$ tem pouco efeito no consumo de memória do ambiente.

Quanto ao efeito de interação entre os fatores avaliados, pode ser identificado pela variação do ângulo entre as retas dos fatores na Figura 5(b). Os resultados apresenta-

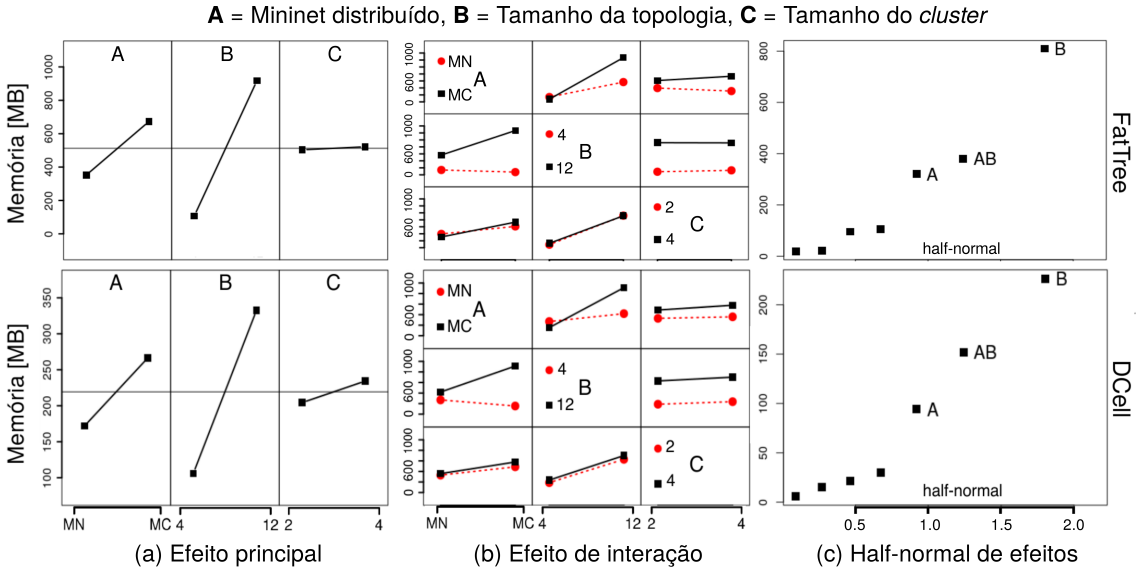

Figure 5. Relação do efeito principal (a), efeito de interação (b) e *half-normal* de efeitos observados durante a experimentação para as topologias FatTree e DCell.

dos no gráfico permitem concluir que a interação entre os fatores $TamanhoTopologia$ e $MininetDistribudo$ causa oscilação no consumo de memória, comportamento não observado de forma relevante na interação dos outros fatores.

Para mensurar o efeito de um fator, ou conjunto deles, sobre o resultado, pode-se fazer uso do gráfico de *half-normal* de efeitos, Figura 5(c). Como exemplo, nele é possível observar que o fator $MininetDistribudo$ causou uma variação de aproximadamente 380MB no consumo de memória, enquanto sua interação com o fator $TamanhoTopologia$ causou um efeito de 400MB e o fator $TamanhoTopologia$ causou um efeito de 880MB de consumo de memória na topologia FatTree.

Com a análise dos resultados apresentados, é possível concluir que, apesar do Mininet Cluster ser a implementação oficial de Mininet distribuído, apresenta comportamento inferior ao MaxiNet, pois, este possui estrutura de gerência de elementos remotos mais inteligente, consumindo menos recursos computacionais do *cluster*. Por outro lado, deve-se considerar também a possibilidade do Mininet Cluster apresentar desempenho superior ao MaxiNet dependendo da topologia ou sistemas emulados durante a experimentação, já que ele suporta conexão direta entre elementos Mininet do tipo *host* criados em nós diferentes do *cluster*, função não suportada pelo MaxiNet, como apresentado na subseção 2.2, obrigando a inclusão de *switches* Mininet a cada novo link entre elementos em diferentes nós.

6. Conclusões e Trabalhos Futuros

Este trabalho apresentou os resultados da análise técnica e avaliação comportamental de soluções de emulação leve e distribuída para experimentação de rede. Para isso, foram apresentados e discutidos a estrutura de implementação de algumas soluções baseadas em virtualização por *container*, identificando características que as tornam mais ou menos indicadas para experimentação de rede. Também foi realizado a análise experimental

das soluções Mininet Cluster e MaxiNet, a fim de gerar dados que representassem seu comportamento quanto ao consumo de recursos computacionais do ambiente.

Os resultados apontaram relevante vantagem do MaxiNet sobre o Mininet Cluster quanto ao consumo de memória RAM, número de processos Linux criados e conexões estabelecidas entre os nós do *cluster*, porém, aponta algumas características que sugerem vantagens do Mininet Cluster em cenários de experimentação de topologias que não fazem uso de tecnologias de SDN.

Como trabalho futuro, é relevante a realização de testes que demonstrem o efeito da variação dos recursos de hardware sobre as soluções analisadas, variando componentes como capacidade de memória, poder de processamento e vazão de rede.

Agradecimentos

O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Código de Financiamento 001. Projeto parcialmente financiado pela FAPESP (Processo nº 2015/24352-9) e pelo Instituto Federal do Tocantins (IFTO). Hermes Senger agradece o apoio da FAPESP (Processo 2018/ 2018/00452-2) e CNPQ (Processo 305032/2015-1).

Referências

Al-Fares, M., Loukissas, A., and Vahdat, A. (2008). A scalable, commodity data center network architecture. In *Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication*, SIGCOMM '08, pages 63–74, New York, NY, USA. ACM.

Beshay, J. D., Francini, A., and Prakash, R. (2015). On the fidelity of single-machine network emulation in linux. In *2015 IEEE 23rd International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 19–22.

Brown, N. (2014). Control groups series. `https://lwn.net/Articles/604609/`. Acessado em 03/09/2018.

Burkard, C. (2014). Cluster edition prototype. `https://github.com/mininet/mininet/wiki/Cluster-Edition-Prototype`. Acessado em 23/06/2018.

Canonical. Linux containers: What's lxc? `https://linuxcontainers.org/lxc/`. Acessado em 20/07/2018.

Canonical. Linux containers: What's lxd? `https://linuxcontainers.org/lxd/`. Acessado em 18/07/2018.

Cao, L., Bu, X., Fahmy, S., and Cao, S. (2017). Towards high fidelity network emulation. In *2017 26th International Conference on Computer Communication and Networks (ICCCN)*, pages 1–11.

Daniels, J. (2009). Server virtualization architecture and implementation. *XRDS*, 16(1):8–12.

Docker (2018). Welcome to the docker cloud docs! `https://docs.docker.com/docker-cloud/`. Acessado em 07/10/2018.

Ganesh, P. I., Hepkin, D. A., Jain, V., Mishra, R., and Rogers, M. D. (2012). Workload migration using on demand remote paging. US Patent 8,200,771.

Graber, S. (2016). Network management with lxd (2.3+). https://stgraber.org/2016/10/27/network-management-with-lxd-2-3/. Acessado em 25/07/2018.

Guo, C., Wu, H., Tan, K., Shi, L., Zhang, Y., and Lu, S. (2008). Dcell: A scalable and fault-tolerant network structure for data centers. In *Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication*, SIGCOMM '08, pages 75–86, New York, NY, USA. ACM.

Huang, T.-Y., Jeyakumar, V., Lantz, B., Feamster, N., Winstein, K., and Sivaraman, A. (2014). Teaching computer networking with mininet. In *ACM SIGCOMM*.

Hykes, S. (2014). Docker 0.9: introducing execution drivers and libcontainer. https://blog.docker.com/2014/03/docker-0-9-introducing-execution-drivers-and-libcontainer/. Acessado em 01/10/2018.

Kerrisk, M. (2013). Namespaces in operation, part 1: namespaces overview. https://lwn.net/Articles/531114/.

Lantz, B. and O'Connor, B. (2015). A mininet-based virtual testbed for distributed sdn development. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 365–366, New York, NY, USA. ACM.

Liu, J., Marcondes, C., Ahmed, M., and Rong, R. (2015). Toward scalable emulation of future internet applications with simulation symbiosis. In *Proceedings of the 19th International Symposium on Distributed Simulation and Real Time Applications*, DS-RT 2015, pages 68–77, Piscataway, NJ, USA. IEEE Press.

Merkel, D. (2014). Docker: Lightweight linux containers for consistent development and deployment. *Linux J.*, 2014(239).

Montgomery, D. C. (2006). *Design and Analysis of Experiments*. John Wiley & Sons, Inc., USA.

Ortiz, J., Londoño, J., and Novillo, F. (2016). Evaluation of performance and scalability of mininet in scenarios with large data centers. In *2016 IEEE Ecuador Technical Chapters Meeting (ETCM)*, pages 1–6.

Wette, P., Dräxler, M., Schwabe, A., Wallaschek, F., Zahraee, M. H., and Karl, H. (2014). Maxinet: Distributed emulation of software-defined networks. In *2014 IFIP Networking Conference*, pages 1–9.

White, B., Lepreau, J., Stoller, L., Ricci, R., Guruprasad, S., Newbold, M., Hibler, M., Barb, C., and Joglekar, A. (2002). An integrated experimental environment for distributed systems and networks. *SIGOPS Oper. Syst. Rev.*, 36(SI):255–270.

Yan, J. and Jin, D. (2015). Vt-mininet: Virtual-time-enabled mininet for scalable and accurate software-define network emulation. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, SOSR '15, pages 27:1–27:7, New York, NY, USA. ACM.

Yan, L. and McKeown, N. (2017). Learning networking by reproducing research results. *SIGCOMM Comput. Commun. Rev.*, 47(2):19–26.