UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE ENGENHARIA DE COMPUTAÇÃO

GUILHERME REX PRETTO

Emprego de Contêineres Linux para provimento de Serviço de Nuvem IaaS em Processadores ARM

Monografia apresentada como requisito parcial para a obtenção do grau de Bacharel em Engenharia de Computação.

Orientador: Prof. Dr. Alexandre Carissimi

Porto Alegre
2016

AGRADECIMENTOS

Agradeço a minha família, que sempre esteve ao meu lado, tanto nos momentos bons, quanto ruins. Agradeço a minha namorada incrível por me suportar. Agradeço a todos os meus professores pelos ensinamentos diários. E meus colegas, amigos do Centro dos Estudantes Universitários de Engenharia e Associação Atlética Acadêmica da Escola de Engenharia que participaram dos momentos de lazer/trabalho durante essa jornada.

RESUMO

Um dos motivos para o aumento da adoção de soluções na nuvem é a vantagem de poder contar com uma infraestrutura que é compatível com suas necessidades. As companhias que usam dessa forma de serviço podem, com facilidade, aumentar ou diminuir o hardware necessário para continuar atendendo de forma adequada aplicações. Esse modelo de negócio, conhecido como "*pay as you go*", possibilita que uma empresa pague apenas pelos recursos utilizados. Sendo um modelo alternativo ao método tradicional, no qual, a companhia mantém sua infraestrutura computacional dentro de um centro de processamento de dados e que, normalmente, é subutilizado durante um determinado período. Este trabalho tem objetivo de analisar uma placa de baixo consumo, Raspberry Pi 3 Model B, como plataforma de hardware para o provimento de serviços de nuvem no modelo IaaS.

Palavras-chave: Computação em Nuvem, virtualização, Linux, contêineres, Raspberry PI, LXC, Docker, IaaS.

Employment of Linux Containers for provision of Cloud Service IaaS in ARM Processors

ABSTRACT

One of the reasons to growth of cloud computing adoption during the recent years is the advantage of usage an infrastructure that can be compatible with their needs. The companies that use this type of service can easily increase or decrease the hardware needed to continuous maintain their applications. This business model known by pay as you go allows that a company pay just the resources used. It is an alternative model to keep a computational infrastructure just to the company, and that normally is underused during a period of time. This work has the objective to analyze a board of low power consume based on ARM processor as a hardware to provide services of cloud computing using an IaaS model.

Keywords: Cloud Computing, virtualization, Containers Linux, Raspberry PI, LXC, Docker, IaaS.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

API	*Application Programming Interface*
AUFS	A*dvanced Multi-layered Unification Filesystem*
AWS	*Amazon Web Services*
BTRFS	*B-tree File System*
CAPEX	*Capital Expenditure*
CGROUPS	*Control Group Config*
CLI	*Command Line Interface*
CPU	*Central Processing Unit*
HPC	*High-Performance Computing*
HTTP	*Hypertext Transfer Protocol*
IEEE	*Institute of Electrical and Electronics Engineers*
IBM	*International Business Machines*
IoT	*Internet of Things*
IP	*Internet Protocol*
IPv4	*Internet Protocol Version 4*
IPv6	*Internet Protocol Version 6*
ISP	*Internet Service Provider*
JVM	*Java Virtual Machine*
LXC	*Linux Containers*
MIT	*Massachusetts Institute of Technology*
MPI	*Message Passing Interface*
NAS	*NAS Parallel Benchmarks*
QoS	*Quality of Service*
SLA	*Service Level Agreement*
SSH	*Secure Shell*
TI	Tecnologia da informação

SUMÁRIO

1 INTRODUÇÃO

Apesar de boa parte do público possuir a ideia de que a computação em nuvem é algo recente, na verdade, é uma ideia antiga. Quase tão antiga quanto o próprio computador. Esse conceito surgiu na década de 1960, a partir das ideias de pioneiros como J. R. C. Licklider (um dos principais influenciadores no desenvolvimento da ARPANET), que imaginava a computação como um sistema distribuído global, e John McCarthy, que definia a computação como um serviço público similar a energia elétrica.

Já o termo "computação em nuvem" foi utilizado pela primeira vez em 1997 pelo professor de sistemas de informação, Ramnath Chellappa[1]. Desde lá, vários avanços em diferentes áreas da computação resultaram na possibilidade de colocar em prática o conceito de uma rede global de oferta de serviços. Entre elas, o crescente acesso a dispositivos eletrônicos (computadores, *tablets*, *smartphones*, etc.) e a Internet de alta velocidade que tornaram possível para usuários acessar suas informações quando e onde quisessem.

Ao mesmo tempo, essa tecnologia é uma vantagem para empresas que transferem seus dados de seus servidores locais para a nuvem. Assim, troca-se gastos em infraestrutura de servidores locais por uma taxa de uso para as empresas que oferecem serviços na nuvem. Os motivos para esse tipo de migração podem ser vários, como redução de gastos com a aquisição de equipamentos. A redução de mão de obra especializada para manutenção dessas infraestruturas; economia de energia, já que a potência consumida pelos equipamentos que antes ficavam dentro da empresa agora fica em *data centers*; e, principalmente, a elasticidade, que é a capacidade do ambiente computacional da nuvem aumentar, ou diminuir, de forma automática os recursos computacionais demandados e provisionados para cada usuário conforme necessário.

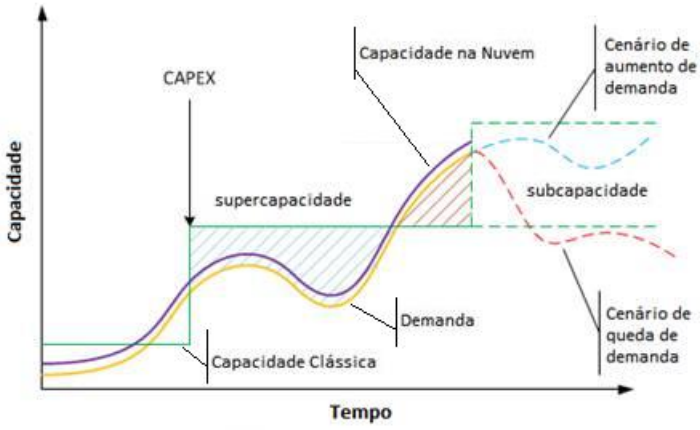
A elasticidade é uma das características mais marcantes da computação em nuvem pelo fato de que aplicações não tem um acesso contínuo de requisições durante um intervalo de tempo. Por exemplo, uma companhia necessita ter poder computacional suficiente para manter suas aplicações funcionando durante picos de uso (Figura 1.1). Vamos dizer que essa companhia vende brinquedos e terá um volume de requisições muito maior durante datas como o dia das crianças e o Natal, do que em outras partes do ano. No modelo clássico, essa empresa necessitará ter capacidade computacional para suportar esses picos de uso, o que

[1] Intermediaries in Cloud-Computing: A New Computing Paradigm Ramnath Chellapp. Apresentado no encontro INFORMS em Dallas, em 1997.

resultará em uma subutilização na maior parte do tempo. Enquanto isso, no modelo de computação em nuvem, é possível variar a capacidade de acordo com a demanda.

A Figura 1.1 tem por objetivo demonstrar de forma gráfica o exemplo anterior, utilizando uma projeção de demanda em função da capacidade necessária para manter o negócio funcionando ao longo do tempo. A curva inferior, identificada por "Demanda", representa a demanda real. Já a curva "Capacidade na Nuvem" tem uma pequena margem sobre a curva de demanda real, e é a capacidade que o modelo em nuvem proporciona. Por outro lado, a curva "Capacidade Clássica" demonstra saltos ao longo do tempo, esses saltos representam o momento onde houve aquisição de equipamentos, sendo possível identificar através do acrônimo CAPEX, sigla da expressão inglesa para *capital expenditure*, que designa o montante de dinheiro despendido na aquisição de bens de capital de uma determinada empresa. Por último, as curvas tracejadas representam previsões futuras de demanda.

Figura 1.1 - Curva de Capacidade e Utilização de Recursos computacionais



Fonte: adaptado de Chades et al. 2010

Outro conceito muito presente na literatura sobre Computação em Nuvem é o conceito de virtualização. A virtualização de recursos está relacionada a maioria das arquiteturas de nuvem. Esse conceito, virtualização, traz uma visão abstrata dos recursos físicos, o que inclui servidores, dispositivos de armazenamento de dados, redes e aplicações, sendo a ideia partilhar recursos físicos e gerenciá-los como um todo. Os gerenciadores de recursos virtualizados podem ser softwares comerciais ou *opensource*.

Atualmente, um movimento que vem ganhando força é a computação verde (*Green Computing*), que é a procura por métodos e práticas para utilizar a energia de forma eficiente a fim de minimizar os efeitos no ambiente. Assim, é possível maximizar a eficiência

energética, que é quantificada pela relação de MFlops/Watt [FENG, 2003]. Esse movimento colabora não somente com questões ecológicas, como também, econômicas.

Uma das formas de *Green Computing* é a utilização de processadores de baixo consumo, como, por exemplo, ARM. Com isso em mente, este trabalho tem como objetivo analisar a viabilidade de uma placa de baixo consumo, Raspberry Pi, que possuiu um processador ARM, para ser utilizada como infraestrutura física para prover serviços de nuvem no modelo IaaS. Entre as tecnologias utilizadas para o projeto estão virtualização, serviço em nuvem, microcontrolador e dispositivos de armazenamento de informações.

Os tipos de tarefas mais comuns de serem colocadas em uma plataforma de Nuvem IaaS são: *data centers* virtuais, que servem para aplicações de diferentes cargas de trabalho; *batch computing*, sendo substitutos ao tradicional método de HPC; e por último, hospedar aplicações de forma individual ou grupo de aplicações relacionadas. Sendo os dois últimos casos, possíveis enfoques onde este trabalho poderia ser utilizado.

1.1 Objetivos do trabalho

Este trabalho tem como objetivo avaliar o uso da Raspberry Pi 3 Model B, devido a placa possuir processador ARM, baixo custo e baixo consumo como hardware básico para o provimento de máquinas virtuais visando oferecer um modelo de IaaS de nuvem computacional, além disso avaliar o Docker como ferramenta de virtualização e gerenciamento de nuvem pelo fato de consumir pouco recurso de hardware. Além disso, será realizada a avaliação de elasticidade e a disponibilidade da infraestrutura.

1.2 Organização do texto

Este trabalho é organizado em 6 capítulos, além desta introdução. O capítulo 2 aborda os principais conceitos de virtualização e de computação em nuvem. Em seguida, o capítulo 3 fornece a solução proposta para prover uma solução IaaS, contendo a arquitetura do sistema. No capítulo 4, são discutidos os detalhes da implementação, tais como softwares e modificações necessárias para atingir o objetivo de prover IaaS usando uma plataforma ARM de baixo custo. Já no capítulo 5, é apresentado uma avaliação do protótipo, contendo as dificuldades encontradas, cenários de teste, resultados obtidos, infraestrutura e metodologia.

Por fim, o capítulo 6 apresenta a conclusão do trabalho, mostrando os principais desafios e contribuições do trabalho, assim como trabalhos futuros.

2 VIRTUALIZAÇÃO E COMPUTAÇÃO EM NUVEM

Os avanços tecnológicos das últimas décadas propiciaram um mundo conectado. A informação trafega de um lado a outro do globo em instantes. Assim, tornaram economias que anteriormente eram locais em globais buscando o máximo de lucratividade em uma atividade para se manter no mercado, pois, o produtor não compete apenas com outros produtores locais, mas do mundo como um todo. A área de tecnologia faz parte dessa regra, e a companhias buscam o máximo de desempenho e utilização de recursos. A computação em nuvem, que é baseada fortemente em virtualização, é um modelo econômico que auxilia nessa eficiência de utilização de recursos, maximizando os recursos computacionais que a empresa necessita em função dos gastos em Tecnologia de Informação (TI).

Assim, este capítulo visa esclarecer os conceitos gerais de virtualização, a seguir serão apresentados seus principais conceitos, diferentes tipos de máquinas virtuais, diferenças entre virtualização total e paravirtualização, virtualização assistida por hardware, e exemplos de ferramentas de virtualização. Além disso, serão apresentados os conceitos de computação em nuvem, discutindo as principais definições do termo, as classificações dos diferentes tipos de serviços que possui, e suas formas de disponibilização.

2.1 Virtualização

A virtualização é um assunto que desperta atenção, apesar de ser um conceito que não é novidade. Pode-se dizer que a ideia da virtualização nasceu com a publicação do artigo "*Time sharing processing in large fast computers*", escrito por Christopher Strachey na Conferência Internacional de Processamento de Informação, realizada em Nova York em 1959 [Strachey, 1959]. Esse artigo tratou do uso da multiprogramação de tempo compartilhado. A partir desse artigo, o MIT desenvolveu o padrão *Compatible Time Sharing System* (CTSS). E, do CTSS, a IBM introduziu o conceito de multiprocessamento nos *mainframes* permitindo que várias CPUs respondessem como uma só, antecipando assim o conceito de virtualização. Esses mesmos *mainframes* introduziram o conceito de memória virtual que possibilitaram o mapeamento da memória real para memória virtual, o que deu início as primeiras formas de virtualização.

Já em 1974, Popek e Goldberg, definiram algumas condições suficientes para uma arquitetura de computador suportar a virtualização do sistema de forma eficiente no artigo "*Formal Requirements for Virtualizable Third Generation Architectures*" [Popek, 1974].

Nesse artigo, estabeleceram três características essenciais para um monitor de máquina virtual (MMV) ou VMM, do inglês, *Virtual Monitor Machine*: equivalência, eficiência e controle de recursos. A equivalência se refere ao fato de que o software executado em uma VMM deve exibir comportamento essencialmente idêntico ao que demonstra durante a execução diretamente no hardware equivalente. Já a eficiência visa que a maioria das instruções devam ser executadas pelo hardware sem intervenção da VMM. Por último, a VMM deve manter controle total dos recursos.

No entanto, é importante ressaltar que virtualização não é um sinônimo de máquina virtual. Há virtualização de rede e de armazenamento, sendo a virtualização de CPU apenas um dos casos. Esse último tipo de virtualização, CPU, tem a ideia de consolidação de servidores como forma de melhorar a eficiência de uso do processador e reduzir o consumo de energia elétrica. Outras formas de virtualização são: virtualização de *desktops*, ambientes de desenvolvimento e testes, e, principalmente, na computação em nuvem.

2.1.1 Tipos de máquinas virtuais

Basicamente, a virtualização consiste na substituição, ou extensão de um recurso, ou interface, por um outro modo que imite seu comportamento. Esse tipo de abordagem é possível devido ao fato de que as interfaces padronizadas entre os componentes do sistema operacional permitem o seu desenvolvimento independente um dos outros.

No entanto, é impossível executar diretamente em um processador Intel/AMD uma aplicação que foi compilada para um processador ARM, pois as instruções em linguagem de máquina do programa não serão compreendidas pelo processador Intel. Da mesma forma, não é possível executar diretamente aplicações para o sistema operacional Windows em um sistema operacional GNU/Linux, pois as chamadas de sistema do programa emissor não serão compreendidas pelo outro sistema operacional [Laureano e Maziero, 2008]. Contudo, é possível solucionar esses problemas com uma camada de virtualização que ofereça a mesma interface para as aplicações, mas que seja construída com os serviços oferecidos pela interface de um outro sistema. Essa camada de software permitirá o acoplamento entre interfaces distintas, a qual, transforma ações de um sistema A em ações equivalentes em um sistema B (Figura 2.1).

Figura 2.1 - Acoplamento entre interfaces distintas

Fonte: Adaptado de Atzori et al. 2010

A virtualização pode ser classificada em três categorias, sendo elas: nível de hardware, nível de sistema operacional e nível de linguagem de programação [Rosenblum, 2004]. Assim, a virtualização em nível de hardware é a camada de virtualização que fica diretamente sobre a máquina física e apresenta para as camadas superiores um hardware abstrato similar ao original. Este tipo corresponde à definição original de máquina virtual dos anos 60.

O hipervisor é a plataforma básica das máquinas virtuais. Entre suas principais funções estão o escalonamento de tarefas, gerência de memória e manutenção do estado da máquina virtual. A qualidade de um hipervisor é definida pelo desempenho e pela escalabilidade. Também é desejável que um hipervisor possua segurança sobre os recursos virtualizados e agilidade na reconfiguração de recursos computacionais, sem a interrupção de operações do servidor de máquinas virtuais. A seguir serão apresentadas as duas formas básicas de hipervisores [Goldberg, 1973] [IBM, 2008].

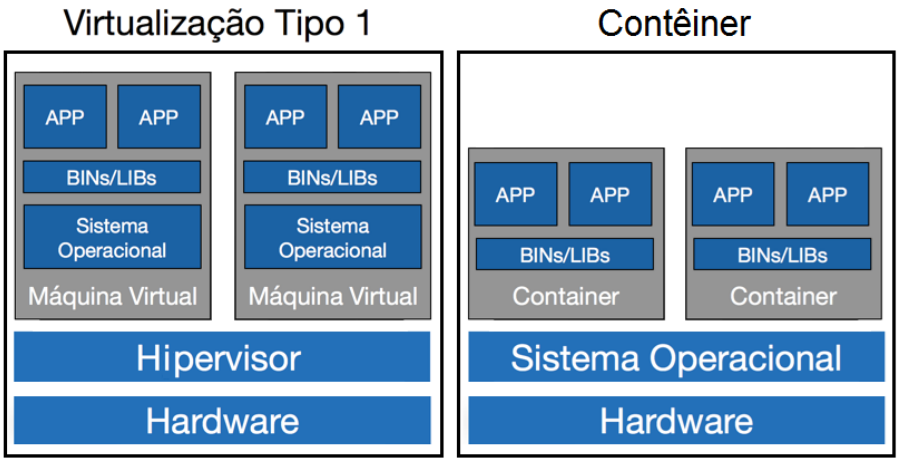
O hipervisor Tipo I, denominado de *bare metal*, nativo ou superior, é executado diretamente sobre o hardware. Um hipervisor nativo compartilha os recursos de hardware entre as máquinas virtuais, de forma que cada máquina virtual acredita que possui exclusividade sobre os recursos de hardware. São exemplos: VMware ESX Server, Xen Server e Microsoft Hyper-V.

Os hipervisores do Tipo II são caracterizados por executar sobre um sistema operacional nativo como se fosse um processo. Nesse caso, o hipervisor oferece uma camada de virtualização composta por um sistema operacional hóspede e por um hardware virtual criado sobre os recursos de hardware que são oferecidos pelo sistema operacional nativo. Exemplos são: VMware player, Virtualbox e Virtual PC.

Já a virtualização em nível de sistema operacional é o mecanismo que permite a criação de partições lógicas em uma plataforma computacional de maneira que cada partição seja vista como uma máquina isolada, compartilhando o mesmo sistema operacional. Assim, essa camada de virtualização é inserida entre sistema operacional e as aplicações. Alguns exemplos dessa forma de abordagem são: Jails, Chroot, Sun VirtualBox, *Linux Containers*, Docker, KVM e SandBox.

Esse tipo de virtualização permite que a partir de um único hospedeiro sejam criadas múltiplas instâncias isoladas de um determinado sistema operacional, sendo possível pensar que é uma maneira de virtualizar aplicações dentro de um servidor Linux. O precursor desse conceito foi o *chroot*, passando pelo comando *jail* (introdução da ideia de segregação de rede e limitação dos acessos de super usuários aos processos), até chegar no *Linux Container* que traz um nível de segurança maior que seus precursores. Esse tipo de virtualização não precisa de um sistema operacional para cada aplicação. Sendo que esse tipo de virtualização demanda menos recursos se comparado com a virtualização tipo 1 (Figura 2.2). Já que não necessita um sistema operacional para cada máquina virtual. Entretanto, dentro de cada contêiner continua existindo uma camada composta por binários (BIN) e bibliotecas especificas LIB. O diretório BIN contém arquivos binários executáveis que armazena alguns comandos básicos do sistema. Enquanto que LIB é um conjunto de rotinas que são utilizadas pelos programas.

Figura 2.2 - Diagrama comparativo entre Virtualização Tipo 1 e Contêiner



Fonte: Autor

Por fim, a virtualização em nível de linguagem de programação em que a camada de virtualização é um programa de aplicação do sistema operacional. O seu objetivo é definir uma máquina abstrata, onde é possível a execução de uma aplicação que foi desenvolvida em

uma linguagem específica de programação de alto nível. O exemplo mais marcante dessa abordagem é a máquina virtual Java (JVM).

Apesar dessas três categorias possuírem diferentes objetivos, elas buscam aspectos em comum: oferecer compatibilidade de software e permitir isolamento entre máquinas virtuais, para que, os softwares não afetem o funcionamento uns dos outros. O cuidado com o projeto dessa camada é primordial, pois a mesma não deve afetar o desempenho das aplicações.

2.1.4 Virtualização total e paravirtualização

No entanto, a implementação de máquinas virtuais é mais complicada que aparenta. Além da preocupação direta com desempenho, existe um problema de como os recursos físicos da máquina são compartilhados entre o sistema nativo e o sistema hóspede sem que ocorra interferência entre eles. O principal problema se deve ao fato de como proceder quando instruções privilegiadas[2] (*System ISA*), do sistema hóspede são executadas [Carissimi, 2009].

A virtualização total é a réplica (virtual) do hardware subjacente de forma que o sistema operacional e as aplicações podem ser executadas como se estivessem diretamente sobre o hardware original. A principal vantagem desse método é que o sistema operacional não necessita de modificação para ser executado sobre o monitor de máquina virtual ou hipervisor.

No entanto, a virtualização total tem três inconvenientes. O primeiro deles é a necessidade de todas instruções executadas pelo sistema hóspede precisarem ser testadas na máquina virtual para saber se as mesmas são instruções privilegiadas. Quando uma instrução é privilegiada, ela deve ser interceptada e emulada no hospedeiro para evitar que a máquina virtual não interfira do sistema nativo, resultando em um processamento oneroso, se o processador nativo não possuir suporte a hardware para virtualização.

Já o segundo inconveniente é relacionado a complexidade em implementar uma máquina virtual que possua o comportamento exato de um dispositivos de E/S. Isso é diretamente relacionado a variedade de dispositivos de E/S que fazem parte de um computador. Esse problema é solucionado provendo a máquina virtual com suporte a uma série de dispositivos genéricos.

---

[2] Instruções Privilegiadas são instruções que se executadas por um programa em modo de usuário causam exceções.

Por fim, o terceiro inconveniente é relacionado com a forma de implementação da gerência de memória. Uma máquina virtual deve pré-alocar uma quantidade de páginas do sistema nativo e emular sobre elas um espaço de endereçamento físico para o sistema hóspede. Outra possibilidade é usar diretamente a gerência de memória do sistema nativo, o que pode ocasionar a necessidade de uma conversão do espaço de endereçamento do sistema hóspede para o sistema nativo, podendo provocar a disputa de recursos com outros eventuais sistemas hóspedes. Essa técnica acaba gerando uma queda de desempenho.

A paravirtualização visa contornar os problemas de desempenho da virtualização total. Essa abordagem exige que sempre que o sistema hóspede for executar uma instrução privilegiada, ele seja modificado para realizar uma chamada a máquina virtual. Assim, todas as instruções privilegiadas (*system ISA*) que são executadas pelo sistema hóspede são alteradas para chamadas à máquina virtual para que ela interprete e emule essas ações de forma adequada. Essa é a principal desvantagem desse método, pois, nem sempre, é possível ou fácil ter acesso ao código fonte para realizar essa alteração. A vantagem é que todas instruções de usuário (*user ISA*) não precisam de alteração e podem ser executadas diretamente no processador nativo. Com a preservação do *user ISA*, todas as aplicações que foram desenvolvidas para o sistema hóspede podem ser executadas sem alterações.

Outra vantagem da paravirtualização se deve ao fato de prover aos sistemas hóspedes acesso aos recursos de hardware a partir de *drivers* instalados no próprio hipervisor. Assim, os sistemas hóspedes tem acesso aos recursos reais da máquina ao invés de recursos genéricos como era o caso da virtualização total. Além disso, a máquina virtual apenas gerencia o compartilhamento do hardware e monitora as áreas de memória e de disco alocadas para cada sistema hóspede. Assim, a gerência de memória do hóspede é capaz de traduzir diretamente as páginas virtuais em quadros físicos, sem a necessidade de uma etapa de conversão.

2.1.5 Virtualização assistida por hardware

A virtualização assistida por hardware utiliza recursos incorporados nas últimas gerações de CPUs da Intel e da AMD. Essas tecnologias são conhecidas como Intel VT e AMD-V [Uhlig, 2005], respectivamente, que oferecem extensões necessárias para executar máquinas virtuais não modificadas sem as desvantagens da emulação de CPU da virtualização total. Além de Intel e AMD, outros fabricantes de hardware têm se preocupado em oferecer suporte a virtualização. Em 2005, a Sun Microsystems incorporou suporte nativo à virtualização em seus processadores da família UltraSPARC [Yen, 2007]. Enquanto que a

IBM propôs uma especificação de interface de hardware nomeada IBM Power ISA 2.04 [IBM, 2007].

Uma das vantagens desse método é que o sistema hóspede tem acesso direto aos recursos disponíveis sem qualquer emulação, ou modificação, o que acaba resultando em um melhor desempenho. Assim, a virtualização assistida por hardware satisfaz os critérios de Popek e Goldberg [Popek e Goldberg, 1974] e proporciona um melhor desempenho porque as instruções privilegiadas são agora capturadas e emuladas diretamente no hardware. A desvantagem desse método é que requer o apoio explícito no CPU do sistema nativo. No entanto, ainda existem processadores que não suportam tal tecnologia como os processadores ARM.

2.1.6 Ferramentas de virtualização: exemplos

Nos últimos anos a virtualização tem proporcionado uma grande mudança no cenário mundial, com alto investimento de empresas que oferecem soluções de virtualização e, graças a essas ferramentas, tem acontecido uma revolução no método de como corporações e usuários domésticos gerenciam suas informações. O mercado conta com uma infinidade de soluções, que podem ser comerciais, gratuitas ou integradas a sistemas operacionais. Em virtude disso, será apresentado a seguir algumas das possibilidades de ferramentas que podem ser adotadas, como Chroot, Jails, Sandbox e *Linux Containers*.

O comando *chroot* foi lançado em 1979 pelo Unix V7 [UNIX, 1979] com a intenção de diferenciar a forma de acesso a diretórios. Assim, evitando com que o usuário fosse capaz de ter acesso à ao diretório raiz (“/” ou *root*). Ao utilizar o comando *chroot* no diretório raiz. Esse diretório não consegue mais acessar arquivos de fora desse diretório, ou seus processos filhos.

O conceito de *chroot* evoluiu para o comando *Jail* (*https://www.freebsd.org/*), no sistema operacional FreeBSD 4, que é um recurso de criação de múltiplos ambientes virtuais nativo ao FreeBSD. Pode-se dizer que o Jail consiste em um diretório, um endereço de rede e um *hostname* para identificar a máquina virtual, que pode ser acessada tanto por console onde está instalada ou por SSH. Apenas arquivos e processos ficam dentro de um *Jail*, o núcleo do sistema operacional em funcionamento é único. Portanto, se comparado a ambientes como *chroot* essa solução economiza processamento gerando apenas um uso maior de memória física de acordo com os processos iniciados na *jail*.

O Sandbox (*www.sandboxie.com*) é uma ferramenta que possibilita a execução de programas e seus processos, sendo possível realizar testes em um ambiente seguro e controlado. Ele torna um ambiente virtual fechado, sendo que as atividades de um aplicativo são executadas normalmente. Contudo, esses aplicativos estarão em uma área com operações limitadas e não afetarão o restante que está no computador, mas fora dessa região. Algumas formas de utilizar esse recurso é através de antivírus como Avast e o Comodo, que possuem opções para ativar esse mecanismo.

Linux Containers (*www.linuxcontainers.org*), ou LXC, é um método de virtualização em nível de sistema operacional que pode executar vários sistemas GNU/Linux isolados em um único hospedeiro de controle. Um LXC fornece um ambiente virtual que possui seus próprios recursos, como CPU, memória, bloco de E/S, rede, etc, que é fornecido por grupos de controle no núcleo do Linux hospedeiro, sendo semelhante ao *chroot*, mas com um nível de segurança maior. Um dos motivos para se usar essa abordagem é a economia de recursos e rapidez na inicialização.

O *Linux Container* foi inicialmente uma ferramenta utilizada por desenvolvedores pela facilidade de criação e realização de teste de aplicações. Além disso, com a consolidação da virtualização, e da expansão da computação em nuvem, surgiu um ambiente propício para a adoção do *Linux Container*. Em virtude dessa técnica, aplicações podem ser portadas de forma direta entre computador de um desenvolvedor para um servidor em produção, ou até mesmo para uma instância virtual em uma nuvem pública.

2.2 Computação em nuvem

Inicialmente o termo *cloud* (nuvem), e sua imagem, eram usados pela telefonia. Posteriormente foi adotado como metáfora pela área de computação, descrevendo a Internet nos diagramas de redes. A imagem da nuvem era usada para indicar algo intangível, mas que era necessário para o funcionamento das redes, tanto que, as linhas desenhadas nos diagramas passavam por dentro da imagem da nuvem ilustrando a Internet somente como um caminho intermediário na transferência de dados [Taurion, 2009].

A ideia fundamental de computação em nuvem é a utilização e o acesso a informação independente de lugar ou plataforma. Dessa forma, a computação em nuvem torna possível a utilização de recursos como redes, servidores, armazenamento, aplicações e serviços, os quais podem ser compartilhados através de um repositório computacional podendo ser rapidamente escalados, em virtude de novas demandas, com mínimo esforço de gerência ou interação do provedor do serviço. O usuário ou aplicação é tarifado de acordo com os recursos que aloca. Esse modelo de negócio é conhecido por "*pay as you go*".

2.2.1 Características Essenciais

Em 2011, o *National Institute of Standards and Technogy (NIST)* definiu várias características para a computação em nuvem [Mell, P., 2011]. Entre as cinco características essenciais está o serviço sob demanda automático (*On-demand self-service*), onde um consumidor deve ser capaz de alocar automaticamente recursos computacionais sem interação humana, como, por exemplo, o armazenamento.

O acesso amplo via rede (*Broad network access*) é outra característica, e se refere à capacidade de acessar os recursos disponíveis através da Internet. Esses recursos devem ser acessíveis por mecanismos padronizados, que permitam seu uso por diferentes dispositivos, como computadores pessoais, *tablets*, *smartphones*, estações de trabalho, etc.

A terceira característica diz respeito ao agrupamento de recursos (*Resource pooling*). Os recursos computacionais do provedor de serviços são agrupados para servir a múltiplos consumidores, sendo os recursos tanto físicos, quanto virtuais, arranjados dinamicamente conforme demanda dos consumidores. Deve haver senso de independência de localização, isso é, o consumidor não tem um controle exato de onde os recursos utilizados se encontram,

mas deve ser possível especificar esse local em alto nível de abstração. Por exemplo, país, unidade federativa ou *data center*.

Outra característica é a elasticidade rápida (*Rapid elasticity*), onde recursos devem ser alocados e liberados permitindo a rápida adaptação à demanda. Para um consumidor, os recursos disponíveis devem parecer ilimitados, sendo possível alocar a quantidade desejada desses recursos a qualquer momento.

A última característica essencial, mas não menos importante, é a de serviços mensurados (*Measured service*). No qual os sistemas em nuvem devem controlar e instanciar recursos de forma automática, disponibilizando mecanismos para contabilizar a utilização de recursos através de um sistema de medida apropriado para o tipo de recurso em uso. Deve ser possível monitorar, controlar e consultar o uso dos recursos, assim provendo transparência para provedor e consumidor do serviço.

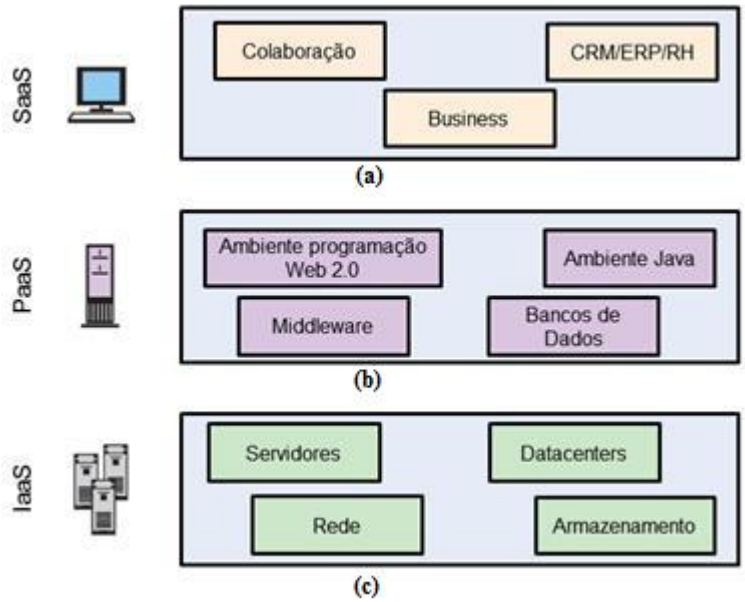
2.2.2 Modelos de Serviço

O NIST [Mell, P., 2011] define ainda três modelos de serviço para computação em nuvem. O primeiro deles é o *Software as a Service (SaaS)*, por vezes referido como *software on-demand*, que é a capacidade de prover ao consumidor o uso de aplicações na infraestrutura da nuvem. Essas aplicações são acessíveis a vários clientes por meio de uma interface como um navegador web ou de um programa específico. O consumidor não tem nenhum controle sobre gerenciamento da infraestrutura da nuvem (rede, servidores, sistema operacional, armazenamento, ou capacidades individuais da aplicação, com à exceção de limitadas opções de configurações de uma aplicação). Exemplos desse tipo de modelo são aplicações de negócios como softwares de contabilidade, gestão, planejamento de recursos empresariais, gerenciamento de conteúdo e serviço (Figura 2.3a). Esse modelo tem sido incorporado na estratégia das maiores empresas de software do mercado. De acordo com estimativa realizada pelo Gartner Inc, em 2019, aproximadamente 28% dos sistemas de gestão de capital humano instalados globalmente serão baseados em SaaS, em 2014 a participação de SaaS nesse nicho era de 13% [GartnerSource2015].

O próximo modelo é *Platform as a Service (PaaS)*, que define a capacidade de prover a implantação na infraestrutura da nuvem de um conteúdo desenvolvido pelo próprio cliente, ou aplicações criadas usando linguagens de programação, bibliotecas, serviços, e ferramentas suportadas pelo provedor. O consumidor não tem qualquer controle sobre a gerência da infraestrutura da nuvem (rede, servidores, sistema operacional, ou armazenamento), mas tem

controle sobre a forma de implementação do aplicativo e, possivelmente, das definições de configuração de ambiente de hospedagem da aplicação. As ofertas PaaS facilitam a implantação de aplicações sem o custo e a complexidade de aquisição e gerenciamento do hardware subjacente, do software e de recursos de provisionamento de hospedagem, proporcionando as facilidades necessárias para suportar o ciclo de vida completo de construção e entrega de aplicações web, além de serviços disponíveis por completo a partir da Internet (Figura 2.3b).

Figura 2.3 - Modelos de serviços para computação em nuvem



Fonte: Adaptado de Zhou et al. 2011

Por último, a definição de *Infrastructure as a Service (IaaS)*, que é a capacidade de prover ao consumidor a possibilidade de dispor processamento, armazenamento, rede, e outros recursos básicos de computação. O consumidor não gerencia ou controla a infraestrutura da nuvem, mas tem controle sobre todas as funcionalidades do sistema operacional, armazenamento e a forma de distribuição da aplicação. O consumidor também pode ter acesso limitado ao controle de componentes de redes, por exemplo, *firewalls*.

O modelo IaaS é o segmento que tem o maior crescimento nos últimos anos. Atualmente, IaaS tem uma participação de 16,2 bilhões em receita anual, sendo projetado que, apenas no ano de 2016, esse mercado cresça 38,4%. Um dos principais fatores para esse crescimento é o reflexo de serviços de tecnologia da informação (TI) legados estarem sendo

migrados para serviços baseados na nuvem, afirma Sid Nag, diretor de pesquisas da Gartner Inc [GartnerSource2016].

2.2.3 Modelos de Implantação

Além, dos modelos de serviço, o NIST [Mell, P., 2011] também definiu modelos para a implantação de nuvens, que são: nuvem pública, nuvem privada, nuvem comunitária e nuvem híbrida. As nuvens públicas têm sua infraestrutura disponibilizada para o público em geral, sendo possível o acesso por qualquer usuário que conheça onde o serviço se localiza. Esse tipo de modelo de implantação não tem restrições de acesso quanto ao gerenciamento de redes. Uma nuvem pública pode ser propriedade, gerenciada, e operada por uma empresa, por uma instituição de ensino, ou por uma organização governamental.

Enquanto isso, nos modelos de nuvem privada, a infraestrutura de nuvem é utilizada exclusivamente por uma organização, podendo ser uma nuvem local, ou remota, que é administrada pela própria empresa ou por terceiros. Esse modelo adota políticas de acesso aos serviços. As técnicas utilizadas para prover tais características podem ser em nível de gerenciamento de redes, configurações dos provedores de serviços e a utilização de autenticação e autorização.

Outro modelo é a nuvem comunitária. Nesse modelo de implantação ocorre o compartilhamento de uma nuvem por diversas organizações que possuem interesses em comum, e possuem requisitos de segurança, de política e de flexibilidade similares. Esse tipo de modelo pode existir localmente, ou remotamente e, geralmente, é administrado por alguma empresa da comunidade ou por terceiros. Um exemplo disso são as nuvens mantidas por serviços públicos governamentais.

Por fim, o modelo de nuvem híbrida, que é composta por dois ou mais estilos de nuvens, podendo ser privada, comunitária ou pública, e que permanecem como entidades únicas, interligadas por uma tecnologia padronizada, ou proprietária, que permite a portabilidade de dados e aplicações.

2.2.4 Gerenciamento da Nuvem

A arquitetura de computação em nuvem é baseada em camadas, onde cada camada trata de uma particularidade da disponibilização de recursos. Cada camada pode ter seu gerenciamento, ou monitoramento, realizado de forma independente das outras camadas,

melhorando a flexibilidade, reuso e escalabilidade com adição ou remoção de recursos computacionais sem afetar as outras camadas [Sousa, 2009]. Na Figura 2.4 é possível ver as camadas referentes à arquitetura da computação em nuvem e suas respectivas associações.

A camada mais de baixo nível é a de infraestrutura física fornecendo os recursos de hardware para as camadas acima. Essa camada fornece flexibilidade e facilidade para agregação de novos recursos à medida que se tornem necessários.

Figura 2.4 - Arquitetura da Computação em Nuvem

Fonte: Adaptado de Vecchiola, Chu & Buyya et al. 2009

A camada de *middleware* é responsável pelo gerenciamento da infraestrutura física e tem por objetivos prover um ambiente de execução apropriado para as aplicações e explorar de maneira eficaz os recursos físicos. Essa camada pode ser dividida em duas subcamadas: uma responsável por garantir o isolamento de processos e aplicações, podendo utilizar tecnologias de virtualização para isso; e outra camada responsável por prover um conjunto de serviços que auxiliam os provedores de serviços comerciais e profissionais - dentre os serviços dessa camada podem ser encontrados negociação de Qualidade de Serviço (QoS), gerenciamento de *Service Level Agreement* (SLA), serviços de cobrança, gerenciamento de requisições, entre outros.

No nível acima da camada de *middleware*, encontra-se a camada responsável por prover suporte para a construção de aplicações e que contém ferramentas, ou ambientes de desenvolvimento. Esses ambientes possuem interfaces Web 2.0, *mashups*, componentes, recursos de programação concorrente e distribuída, suporte a *workflows*, bibliotecas de programação e linguagens de programação. Essa camada de desenvolvimento não é utilizada pelos usuários finais, e sim, pelos usuários mais experientes, aqueles que desenvolvem as soluções para computação em nuvem. Essa camada de *middleware* no nível de usuário constitui o ponto de acesso das aplicações à infraestrutura da nuvem.

Por fim, encontra-se a camada das aplicações de computação em nuvem. Essa camada é de interesse do usuário, pois é por meio dela que eles utilizam os aplicativos. As camadas abaixo desta são responsáveis pelas características de escalabilidade, disponibilidade, ilusão de recursos infinitos e alto desempenho.

2.3 Serviços de Nuvens e Ferramentas de Gerenciamento

Atualmente, boa parte dos dados de um usuário estão armazenados em serviços de nuvem. Nomes como Google Drive, Apprenda, Amazon, Azure, entre outros, tantos serviços, fazem parte do dia a dia, possibilitando, desde, o armazenamento de informação, editores de texto *online*, até o provisionamento de máquinas virtuais. Além disso, normalmente, as companhias oferecem aos usuários contas gratuitas com funcionalidades limitadas, e, se for necessário, o cliente pode contratar outros serviços pagos.

O Google Drive (*www.google.com/drive*) é um serviço que possibilita ao dono de uma conta armazenar qualquer tipo de arquivo. Assim que é criada uma conta, o cliente já possui 15GB disponíveis para armazenamento. Esse espaço pode ser utilizado pelo Google Drive, o Gmail e o Google Fotos, sendo que a qualquer momento é possível contratar um plano de armazenamento com capacidade maior.

A Apprenda (*www.apprenda.com*) é uma companhia que prove plataforma como serviço para ajudar companhias a criar, atualizar e gerenciar aplicações baseadas em nuvens públicas e privadas. Ela foi fundada em 2007, em Clifton Park, New York e seu publico alvo são desenvolvedores que trabalham para companhias como bancos, seguradoras e provedores de saúde.

Já a Amazon (*www.aws.amazon.com*), oferece um nível gratuito do Amazon Web Services (AWS), onde o usuário pode entrar em contato com os serviços da Nuvem AWS. Essa gratuidade, para alguns serviços, é válida por 12 meses a partir do momento da criação

da conta. AWS oferece serviços como armazenamento e distribuição de conteúdo, banco de dados, ferramentas de desenvolvedor e gerenciamento, entre outros.

O Azure (*www.microsoft.com/azure*) da Microsoft segue a mesma linha do Amazon. Ao criar sua conta, o usuário tem R$ 750,00 em créditos, os quais pode usar para provisionar máquinas virtuais, bancos de dados SQL, ou espaço de armazenamento. Entre os serviços mais populares do Azure estão: provisionamento de máquinas virtuais, App Service, armazenamento em nuvem, *Active Directory* e backup de servidores. Sendo esses dados extraídos do site oficial[3], durante o último trimestre de 2016.

O gerenciamento é um dos elementos mais importantes para o modelo de serviço IaaS em ferramentas de administração de nuvem. Elas oferecem aos clientes recursos de infraestrutura sob demanda e controlando questões administrativas da nuvem. Assim, as camadas de PaaS e SaaS (camadas superiores) são extremamente dependentes das ferramentas do modelo IaaS. Existem várias ferramentas destinadas ao gerenciamento de infraestrutura em nuvem, sendo que algumas delas são de código aberto. Por outro lado, em demandas específicas podem ser usadas ferramentas proprietárias, que são pagas e oferecem um gerenciamento completo da nuvem [Shroff, 2010]. A seguir são comentados características específicas de algumas ferramentas de código aberto como CloudStack, Eucalyptus, OpenNebula e OpenStack.

O CloudStack [CloudStack, 2016] é uma ferramenta de gerenciamento de IaaS de código aberto. É possível utilizá-la em nuvens privadas, públicas e híbridas. Essa ferramenta possui um número considerável de usuários, sendo uma comunidade ativa e que tem diversos recursos em desenvolvimento. O CloudStack oferece APIs que possibilitam ao administrador e usuários de IaaS gerenciar a infraestrutura computacional. Sua interface gráfica é organizada e fácil de utilizar.

O Eucalyptus (*www.open.eucalyptus.com*) é um software de código aberto baseado em GNU/Linux. Eucalyptus é o acrônimo de *Elastic Utility Computing Architecture Linking Your Programs to Useful Systems*. Foi desenvolvido para suportar nuvens privadas e híbridas e em 2014 foi adquirido pela HPE [Cocozza, 2015].

O OpenNebula (*www.opennebula.org*) é uma ferramenta de gerenciamento de *IaaS* de código aberto para ambientes de nuvem privada e pública. É implantada baseando-se nos modelos clássicos de *cluster*. Sendo um nodo da infraestrutura usado como mestre, o qual executa os principais serviços da ferramenta e gerencia os demais nodos que são seus

[3] https://azure.microsoft.com/

escravos. Além do núcleo principal da ferramenta OpenNebula (ONED), que controla parte dos componentes, existe o OneFlow e OneGate. Enquanto que a comunicação entre os nodos é feita através do protocolo SSH e do NFS, o OpenNebula permite o provisionamento de instâncias na infraestrutura física [OpenNebula2016].

O OpenStack (*www.openstack.org*) que é uma ferramenta de gerenciamento de *IaaS* de código aberto que nasceu de um projeto entre Rackspace e a NASA. Atualmente, o OpenStack é usado para construção de nuvens *IaaS* públicas e privadas. É uma ferramenta largamente utilizada por grandes empresas que investem em desenvolvimento de novos recursos [OpenStack2016]. A ferramenta trabalha com um conjunto de componentes, onde é possível implementar redes virtuais (NEUTRON), discos virtuais no formato LVM (Cinder), controlar a camada de virtualização em conjunto com os virtualizadores (Nova), e oferecer serviços para bancos de dados (Trove) [Loschwitz, M. 2014].

O Docker (*www.docker.com*) surgiu de um projeto *open source* da empresa DotCloud, que é uma empresa de *PaaS*. O Docker foi ganhando prestigio pela maneira de gerenciar os contêineres no ambiente Linux. Após um ano da primeira versão, lançou sua própria biblioteca assumindo diretamente o controle dos *drivers* com o núcleo do sistema. Essa opção agradou os principais mantenedores de Linux e Contêineres, culminando na participação da Canonical (Ubuntu), a Parallels e a RedHat para o desenvolvimento do Docker.

A partir de uma máquina hospedeiro é possível executar um ou mais Dockers, sendo que todos os Dockers de um *host* executam no mesmo núcleo, porém, logicamente isolados uns dos outros usando as tecnologias LXC, Aufs e Btrfs. O Aufs é um serviço de arquivos que implementa uma união para montar sistemas de arquivos Linux, distribuições como Debian, Ubuntu e Gentoo utilizam Aufs. Já o Btrfs é um sistema de arquivos inicialmente desenvolvido pela Oracle Corporation em 2007, sendo Chris Mason seu principal desenvolvedor.

A plataforma Docker disponibiliza um serviço de nuvem para armazenar e compartilhar imagens prontas, que podem ser criadas pela companhia responsável pelo Docker, ou por qualquer pessoa interessada em colaborar. Uma pessoa registrada nessa plataforma pode criar um número ilimitado de imagens públicas e uma imagem privada na conta gratuita. Atualmente, Docker é suportado por Windows Server, MAC, Windows, Linux, AWS e Azure. Além de ter uma comunidade que tem crescido muito nos últimos anos, na Tabela 1 é possível encontrar dados sobre a atividade da comunidade Docker.

Tabela 1 – Atividade da comunidade Docker

Atividade da comunidade Docker	
Download de Contêineres	Superior a 1.2 Milhões
Desenvolvedores com treinamento	Superior a 45 mil
Repositórios públicos	Superior a 8 mil
Cidades com encontros regulares	70
Integração	OpenStack, RHEL, Ubuntu, Chef, Puppet, Salt ++

Fonte: IBM Inter Connect 2015

2.4 Considerações finais

Neste capítulo, foram apresentados os principais conceitos de virtualização e computação em nuvem. Foram abordadas as principais tecnologias envolvidas e os domínios nos quais são úteis para computação em nuvem. Além, da exposição de alguns produtos utilizados a partir de serviços na nuvem e que estão presentes no nosso dia a dia. Apesar de ser uma tecnologia que veio para ficar, ainda existem desafios na área. Entre eles, o consumo de energia que *data centers* utilizam. Não apenas com o processamento, mas, também, em refrigeração. Em virtude disso, e com os conceitos vistos neste capítulo são empregados na sequência para definir uma proposta de arquitetura baseada em placas de baixo consumo de potência para criar uma infraestrutura de hardware para computação em nuvem.

3 SERVIÇO DE NUVEM EM PLACA DE BAIXO CONSUMO

Neste capitulo será apresentada a proposta do trabalho, que é analisar a viabilidade de usar uma placa de baixo consumo para disponibilizar uma infraestrutura de hardware para computação em nuvem provendo suporte a IaaS. Este projeto tem como ideia principal a utilização de uma placa Raspberry Pi 3 Model B e softwares livres para sua construção.

Atualmente, no mercado, existem dezenas de diferentes placas que são utilizadas em projetos de sistemas embarcados, como a Intel Galileo, a Cubietruck, a Beaglebone, a Raspberry Pi, entre outras. Tipicamente, essas placas são utilizadas para desenvolvimento de projetos em sistemas embarcados. O principal motivo para a utilização da Raspberry Pi 3 foi seu baixo custo e a facilidade de aquisição no mercado.

3.1 Arquitetura geral do sistema

A proposta deste trabalho é implementar e analisar a viabilidade de usar o Raspberry Pi 3 para disponibilizar uma infraestrutura de hardware para computação em nuvem. Esse projeto contará basicamente com 3 atores: infraestrutura de hardware, um *front-end*, como gerenciador de recursos e o usuário.

A infraestrutura de hardware, será responsável por executar aplicações e armazenar informações. Nesse nível existirá um software que, através de troca de mensagens com o gerenciador, fará a alocação dos recursos. Para a execução dessas aplicações são criados máquinas virtuais, e espaços de armazenamento virtuais para guardar os dados.

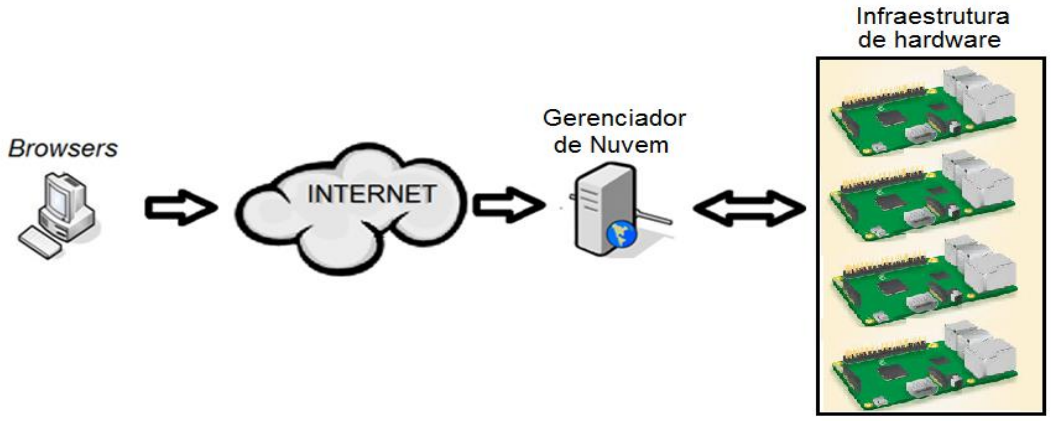
Para a construção do *cluster* foram utilizadas cinco placas Raspberry, é possível visualizar tal arquitetura na Figura 3.1. Nessa arquitetura o *front-end* representa o nó servidor e a infraestrutura de rede são nós clientes.

O nó servidor funciona como um *front-end*, é a partir dele que são executados interações com o *cluster*. Esse nó possui um servidor de arquivos NFS permitindo a distribuição de arquivos entre nós clientes. Além de auxiliar na configuração dos nós clientes através da utilização de conexão SSH. A distribuição Linux utilizada é a Raspbian-Jessie, com *kernel* versão 4.4.15-v7+.

Os nós clientes utilizam um cliente *NFS* para acessar o servidor *NFS* presente no nó servidor. Também possui conexão SSH com o nó servidor, que é utilizada, principalmente, para comunicação durante a execução de aplicações.

Além disso, todos os nós possuem sistema operacional da distribuição Raspbian, LXC, Docker e os demais softwares instalados em um cartão microSD de 32 GB por placa.

Figura 3.1 – Arquitetura Proposta



Fonte: Autor

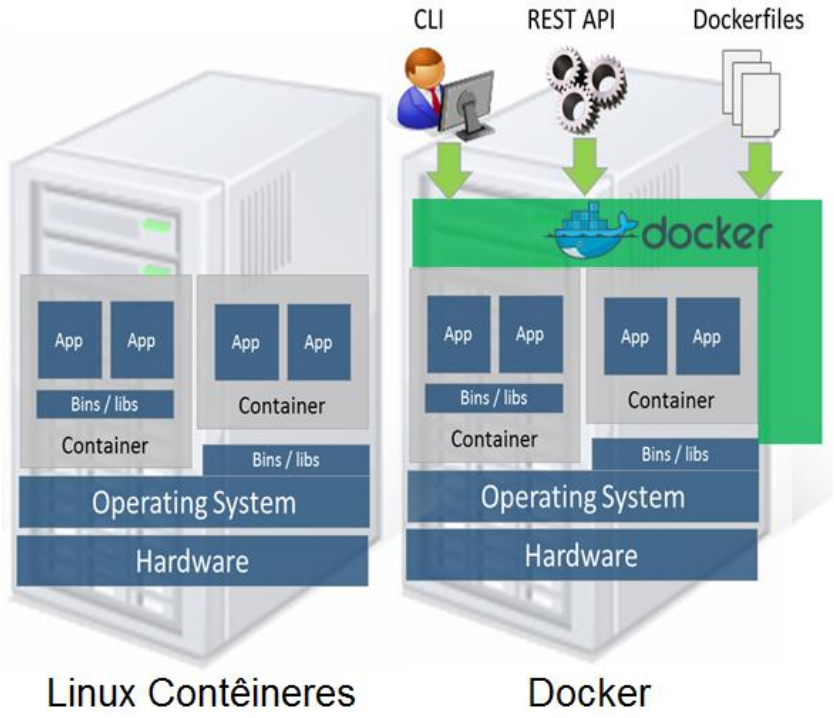
Um dos pontos chaves do projeto foi a modificação e recompilação do *kernel* do Linux pois, sem essa etapa, não seria possível a utilização do LXC na Raspberry, pois existem pacotes necessários do *kernel* que não vem habilitados nessa distribuição e serão detalhados no capítulo a seguir. Uma das razões da utilização de LXC como *back-end* da infraestrutura é devido ao fato de ser uma alternativa leve para virtualização se comparada a hipervisores "tradicionais", como Xen ou KVM. Devido à nossa infraestrutura ter quantidade de recursos limitados se comparada com infraestruturas de grande porte, optou-se por uma tecnologia que tem uma menor sobrecarga nos recursos de hardware. Isso ocorre pelo fato de que o LXC, através do compartilhamento de certas partes do núcleo do hospedeiro, tem um menor sobrecusto.

Já o portal, ou *front-end*, é um servidor que executa um gerenciador de nuvem. Esse gerenciador é implementado por metadados e indexa serviços sobre a infraestrutura da nuvem. O objetivo dessa camada é coletar a entrada do usuário e processá-la para uma forma que o *back-end* possa utilizá-la. Assim, diferentes aplicações podem acessar os mesmos dados, além do que, as aplicações podem ser alteradas sem se importarem com a estrutura e tipo dos dados.

O gerenciador de nuvem é realizado através da plataforma Docker e é uma camada que realiza a interface com o LXC. Essa plataforma é utilizada por desenvolvedores e administradores de sistemas.

O Docker tem por objetivo a criação e administração de ambientes isolados e foi utilizado pela possibilidade de empacotamento de aplicações dentro de contêineres. Isso reduz o tempo de *deployment*, pois não é necessário nenhum tipo de configuração adicional no ambiente para o correto funcionamento do serviço. Então, a partir de uma imagem de contêiner, é possível replicá-la inúmeras vezes. O Docker é uma plataforma Open Source escrito em Go[4].

Figura 3.2 – Recursos que o Docker agrega ao LXC



Fonte: Adaptado Boden Russel et al 2014

Na Figura 3.2, é possível ver o que o Docker agrega ao LXC. Onde, além das funcionalidades do LXC, o Docker possui um CLI, Rest API e Dockerfiles. Ele trabalha utilizando o modo cliente e servidor. Assim, toda a comunicação entre o Docker Daemon e Docker Client é realizada através de API. Na plataforma do Docker também é utilizada a biblioteca *libcontainer*, que realiza a comunicação entre o Docker Daemon e o *back-end* utilizado, sendo ela a responsável pela criação e delimitação dos limites de recursos de cada contêiner.

[4] Linguagem criada pelo Google para suprir necessidades cada vez maiores por computação concorrente e velocidade de processamento. Material disponível em https://golang.org/.

Além disso, o Docker possui nativamente um modo denominado de *Swarm*. Utilizando esse recurso é possível conectar duas ou mais placas Raspberry Pi para elas trabalharem em conjunto. Para iniciar um *cluster* de placas, é necessário escolher um nodo para ser o mestre. Após realizada essa escolha, outras placas podem ser integradas ao *cluster*. Existem duas opções ao ingressar uma nova máquina ao *cluster*, como um novo nodo mestre secundário ou um novo nodo escravo. Novos nodos mestres podem enviar mensagens de gerenciamento para o *cluster*, assim como o mestre primário realiza. Já nodos escravos somente executam contêineres.

Outra vantagem da utilização do modo *Swarm*, é que ele disponibiliza um gerenciamento automático de contêineres sobre a infraestrutura de hardware. Assim, ao disponibilizar um serviço o modo *Swarm* automaticamente faz o *deployment* dos contêineres. O critério para escalonamento de contêineres sobre a arquitetura, é feito analisando a carga de cada um dos nodos disponíveis na infraestrutura. Com isso, tenta manter uma taxa de recursos utilizados por nodo da forma mais igualitária possível. Por exemplo, ao definir um serviço com *n* contêineres sobre uma infraestrutura com *m* máquinas físicas, o *Docker Swarm* vai se encarregar de distribuir esses contêineres, de forma que sejam alocados *n/m* contêineres por máquina. Além disso, conforme contêineres são inicializados e destruídos é possível que aconteça balanceamento de carga ao longo do tempo.

Através de um navegador o usuário pode acessar o portal e solicitar os recursos na nuvem. A partir desse navegador é possível criar, parar, executar serviços na nuvem, e realizar qualquer tipo de gerenciamento. Com isso, disponibilizando esses serviços na nuvem para os clientes, que podem acessá-los de qualquer lugar.

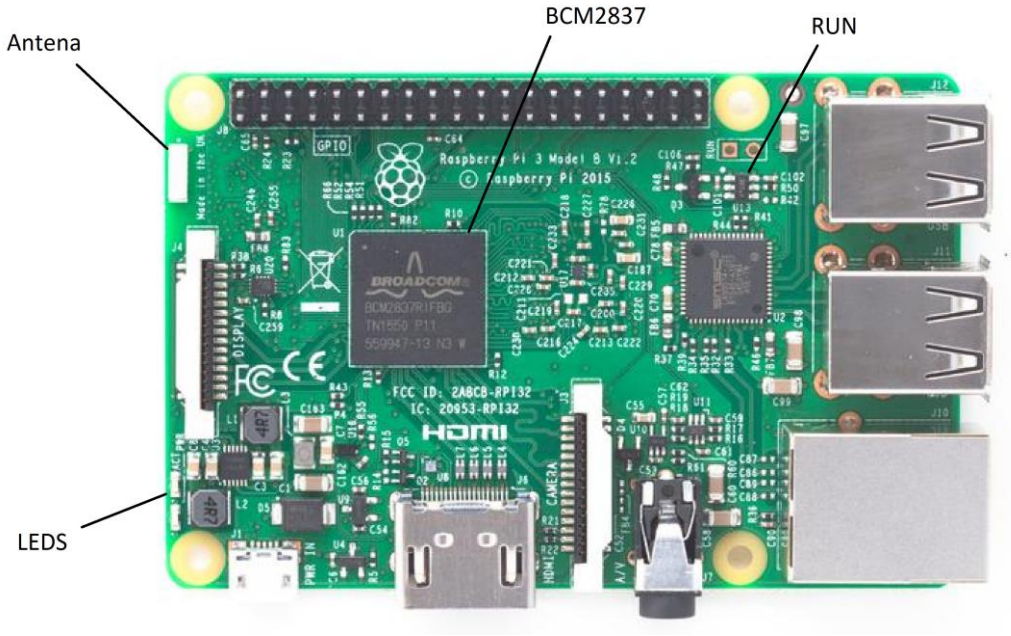
O objetivo final deste trabalho é ter um *cluster* de placas Raspberry Pi, que se comportam como uma infraestrutura de hardware para oferecer suporte à serviços de computação em nuvem. Neste projeto serão utilizadas cinco Raspberry Pi como prova de conceito de que é possível oferecer IaaS com placas de baixo consumo de energia. Além disso, a solução pretendida será composta apenas por software livre.

3.2 Raspberry Pi 3 – Model B

A Raspberry Pi 3 é a terceira geração de placas Raspberry. Seu lançamento foi em fevereiro de 2016 e entrou no mercado para substituir a Raspberry Pi 2 – Modelo B. Algumas melhorias desse modelo em comparação com o anterior, são relacionadas ao novo processador

de 64 bits e a interface de comunicação *wireless*. Existem dois modelos no mercado a Raspberry Pi 3 modelo A+ e Raspberry Pi 3 modelo B.

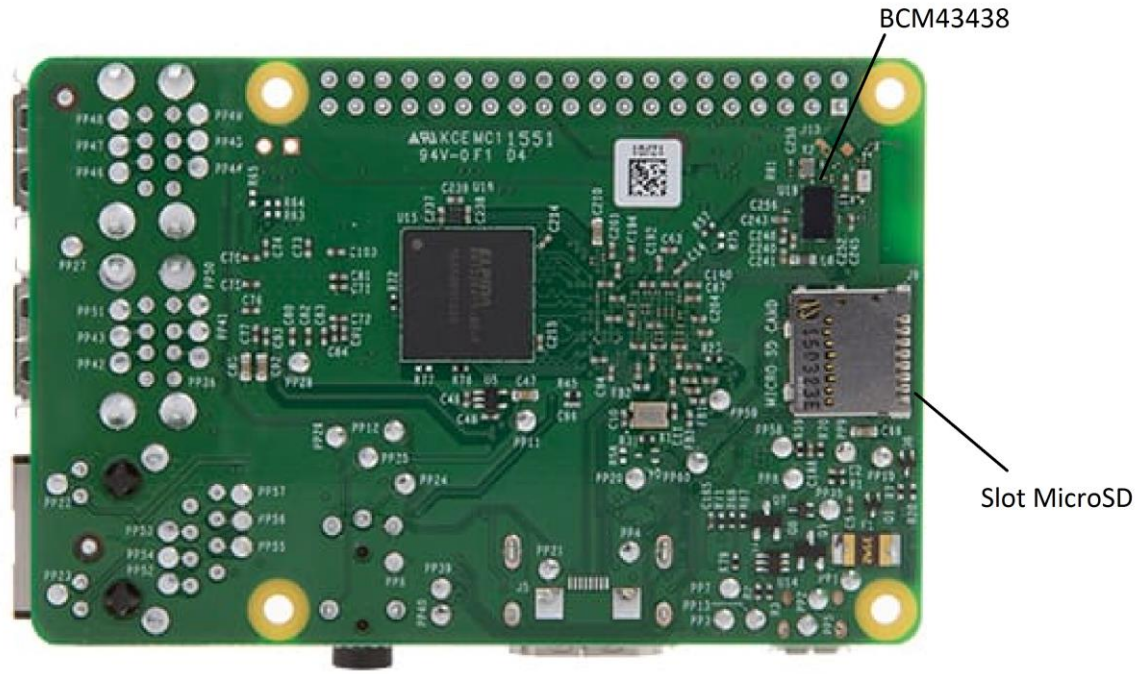
Figura 3.4 - Visão superior da Raspberry Pi 3



Fonte: Autor

Este projeto utiliza uma placa modelo B que possui um SoC, Broadcom BCM2837 (Figura 3.4), que inclui um processador quad-core de 64 bits ARM Cortex-A53 de 1.2GHz. Segundo a The Magpi [Magpi, 2016], esse SoC foi desenvolvido especialmente para esse novo modelo da Raspberry Pi. Além disso, conta com uma memória RAM de 1 Gigabyte LPDDR2 de 900 MHz e GPU Broadcom VideoCore IV que funciona em 400MHz. Esse modelo possui conexão de rede 10/100 Ethernet. Também, existe a possibilidade de usar a interface de comunicação *wireless* 802.11n, tudo graças ao chip BCM43438 (Figura 3.5). Além disso, é possível utilizar um receptor de FM, que vem desabilitado por padrão, Bluetooth 4.1 e Bluetooth Low Energy.

Figura 3.5 - Visão inferior Raspberry Pi 3



Fonte: Autor

A nova placa, Raspberry Pi 3, teve uma modificação em relação à anterior relacionada ao *slot* microSD. Anteriormente, o *slot* era do tipo *push-push*, que é um sistema onde é necessário empurrar o microSD para ele ser liberado. No entanto, isso fazia com que o cartão saísse do *slot* de forma indevida, sendo substituído por um tipo *push-pull*, onde ao inserir microSD o usuário deve empurrar o cartão e deve puxá-lo para retirar. O dispositivo reconhece cartões MicroSD de até 32 Gigabytes de memória. É nesse cartão que fica armazenado a imagem do sistema operacional e demais arquivos.

Outras características dessa placa são a existência de 4 portas USB 2.0, 40 pinos de GPIO, interface com câmera 15-pin MIPI e interface com *display* DSI 15 PIN/HDMI Out/Composite RCA. Sendo necessária a utilização de uma fonte de 2.5 A. O consumo total de energia com uma versão Raspbian padrão instalado e com HDMI e LEDS desligados, é de 1.2 Watts.

A empresa tem feito melhorias no hardware da Raspberry, principalmente, com base no fórum oficial de discussões[5], onde usuários discutem sobre novas funcionalidades e capacidades da placa. Por sua vez, a comunidade e empresas como Ubuntu e Microsoft, tem

[5] Acessível pelo endereço www.raspberrypi.org/forums/.

ajudado no desenvolvimento de sistemas operacionais, especialmente construídos para a Raspberry Pi. Os usuários da Raspberry Pi tem a possibilidade de instalar diferentes sistemas operacionais em suas placas, tudo a partir de um microSD de até 32 Gigabytes. Existe um sistema operacional oficial, o Raspbian[6], que é construído com base na distribuição Debian. Para usuários iniciantes é recomendado a instalação através do NOOBS, que é uma ferramenta que auxilia o usuário a instalar sistemas operacionais. Apesar dessa ferramenta já possuir o Raspbian pré-instalado, é possível instalar um sistema operacional diferente a partir de uma lista de seleção, que automaticamente realiza o *download* de uma imagem e instala a partir da Internet o sistema operacional desejado.

Também estão disponíveis no site oficial, na seção de *downloads* outras distribuições de sistemas operacionais. Por exemplo, Ubuntu Mate, que é baseada no Ubuntu armhf, necessita no mínimo um microSD de 8 GB. O Windows 10 IoT Core, que é uma versão de sistema operacional especialmente construída para ser utilizada em dispositivos com hardware integrados em um única placa. O projeto PiNet, que é um sistema operacional baseado em rede, onde usuários podem se conectar em qualquer Raspberry participante da rede e compartilhar pastas.

3.3 Considerações Gerais

Neste capitulo foram apresentados a arquitetura geral do sistema, e os softwares instalados nessa arquitetura. Também foram apresentado recursos físicos e sistemas operacionais que a placa Raspberry Pi possui.

É possível constatar que há possibilidade de utilizar-se uma placa de baixo consumo para oferecer serviços de nuvem. Graças a utilização de virtualização em nível de sistema operacional, o sobrecusto com cada instância é reduzido consideravelmente. Isso acontece pela forma como os contêineres são construídos se comparados a máquinas virtuais tradicionais, onde para cada máquina virtual existe um sistema operacional completo instalado.

[6] Site oficial do sistema operacional Raspbian, www.raspbian.org/.

4 IMPLEMENTAÇÃO

Baseado no que foi discutido no Capítulo 3, a arquitetura do sistema possui tanto componentes de hardware, quanto de software. Entre os componentes de hardware estão 5 placas Raspberry Pi 3, que, como visto anteriormente, é uma placa de baixo consumo, mas possui recursos de processamento e memória adequados para serem testados neste projeto. Também é utilizado um *switch* de 8 portas modelo TL-SG108 da empresa TP-Link, onde as placas Raspberry Pi 3 foram conectadas através de cabos RJ45.

O sistema operacional que foi utilizado como base para o desenvolvimento do projeto é uma imagem do Raspbian-Jessie. No entanto, o *kernel* do sistema operacional foi reconstruído devido a certas funcionalidades que não são habilitadas e que são necessárias para o funcionamento do LXC. Essas alterações serão discutidas nas próximas seções.

Com os pré-requisitos do núcleo habilitados para o correto funcionamento do LXC, foi possível utilizar o mecanismo Docker, que é uma forma de empacotamento de infraestrutura e utiliza a tecnologia LXC como base. Ela é portátil e simples, proporcionando que múltiplas instâncias de máquinas virtuais sejam executadas em um único hospedeiro.

4.1 Máquinas virtuais baseadas em contêineres

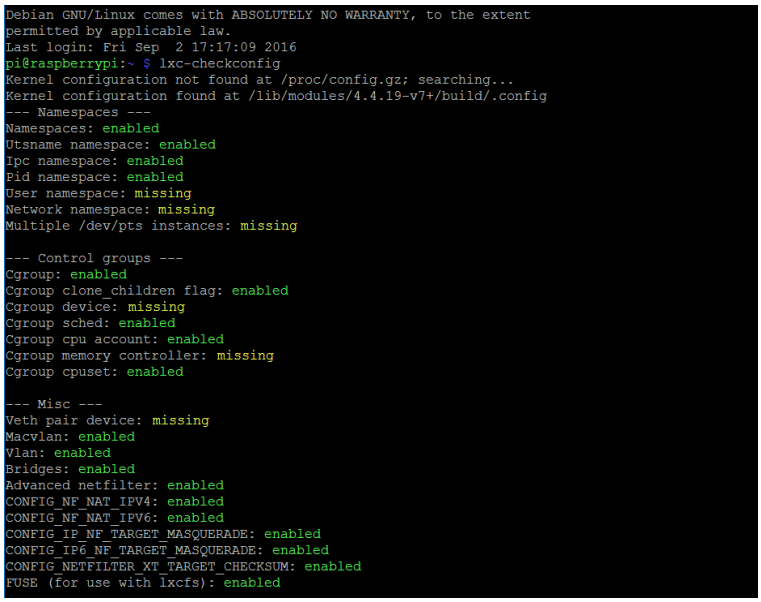
Basicamente, uma máquina virtual baseada em contêineres é, um método de virtualização em nível de sistema operacional que pode executar diversos sistemas GNU/Linux isolados em um único hospedeiro. O LXC oferece um ambiente virtual utilizando *cgroups*, que é um recurso do núcleo do Linux do hospedeiro que permite delimitar regiões de espaço dentro do Linux. Assim, é possível dizer que cada uma dessas regiões delimitadas é um contêiner. Cada um deles possui sua própria CPU, memória, bloco E/S, rede, espaço em disco, etc, sendo uma solução semelhante a um *chroot*, mas oferecendo um isolamento maior.

Os Grupos de Controle, ou *cgroups* (*control group config)*, é um recurso do núcleo que permite limitar e isolar o uso de recursos (CPU, memória, rede) entre usuários definidos por uma coleção de processos. O *cgroups* é utilizado por administradores de sistema, que podem ter um melhor controle sobre alocação, prioridades, gerenciamento e recursos de monitoramento do sistema. Com isso, os recursos de hardware podem ser divididos de forma inteligente entre processos e usuários, elevando a eficiência geral do sistema.

O *cgroups* é um pré-requisito para a configuração e o bom funcionamento do LXC em um sistema operacional. No entanto, esse recurso não vem habilitado com todas as

funcionalidades necessárias na imagem inicial que foi utilizada neste trabalho, a 2016-05-27-raspbian-jessie. Pode-se conferir na Figura 4.1 os recursos que vem habilitados inicialmente.

Figura 4.1 – Funcionalidades do núcleo antes de sua modificação



Fonte: autor

Em virtude disso, foi necessário habilitar sete pacotes do núcleo do sistema. O primeiro é o *Control Group Support* para controle dos recursos de memória, juntamente com as três opções hierarquicamente abaixo dele: *Memory Resource Controller Swap Extension*, *Memory Resource Controller Swap Extension enabled* e *Memory Resource Controller Kernel Memory accounting*, que são recursos de controle de *swap* e memória. Ainda deve ser habilitado o *cpuset support* que, basicamente, seleciona grupos para *cores* específicos e evita com que processos fiquem transitando entre *cores*.

Adicionalmente, foi necessário habilitar duas opções de *drivers* do dispositivo: o *Support multiple instances of devpts* e o *Virtual ethernet pair device*. O primeiro é habilitado para que cada contêiner possa alocar pseudo terminais escravos (*pts*) de forma independente. Um *pts* é recurso que prove aos processos uma interface idêntica a um terminal real. O segundo é um *driver* que permite que túneis sejam criados para comunicação entre *network namespaces*, que permite aos processos ter uma visão própria dos recursos disponíveis no sistema, limitando o que eles tem acesso.

O processo de modificação e compilação foi realizado a partir da própria Raspberry Pi 3, já executando o sistema operacional Raspbian. Inicialmente, foi instalado o sistema operacional na placa, que é um processo relativamente simples. A partir da imagem do

Raspbian-jessie, mencionada anteriormente, foi possível, utilizando programas como o *rawrite32* (que é uma ferramenta para escrever uma imagem em um disco), instalá-la em um microSD. Essa imagem já vem pronta para ser executada diretamente na placa e, após escrever a imagem no cartão microSD, basta inseri-la no *slot* microSD da placa.

O acesso à placa foi realizado através de SSH, utilizando o programa Putty[7]. Esse método foi utilizado pelo fato de já vir habilitado no Raspbian. Outra possibilidade seria conectar um monitor, teclado e mouse.

É preciso uma série de passos para habilitar os recursos necessários para o funcionamento do LXC. Foram realizados, respectivamente: o *download* do *firmware* e módulos, preparação para a construção do núcleo, modificação das opções do *kernel* e a montagem do novo núcleo.

Inicialmente, utilizando Git, foi clonado o *firmware* da Raspberry Pi, do endereço *git://github.com/raspberrypi/firmware.git*. Depois disso, os arquivos foram copiados para os diretórios */boot* e */lib/modules*. Nesse ponto, é necessário aumentar o tamanho do arquivo de *swap* de 100 Megabytes, que é o padrão, para 500 Megabytes. Isso é feito através da alteração do parâmetro de tamanho do *swap* no arquivo */etc/dphys-swapfile*. Após isso, é necessário utilizar o comando *sudo dphys-swapfile setup*, que realizará o ajuste no arquivo de *swap*, bastando reinicializar o sistema operacional para que as mudanças entrem em ação. Sem essa modificação durante a preparação para compilar o núcleo, o processo esgotará toda a memória RAM e abortará durante o processo de gerar o núcleo.

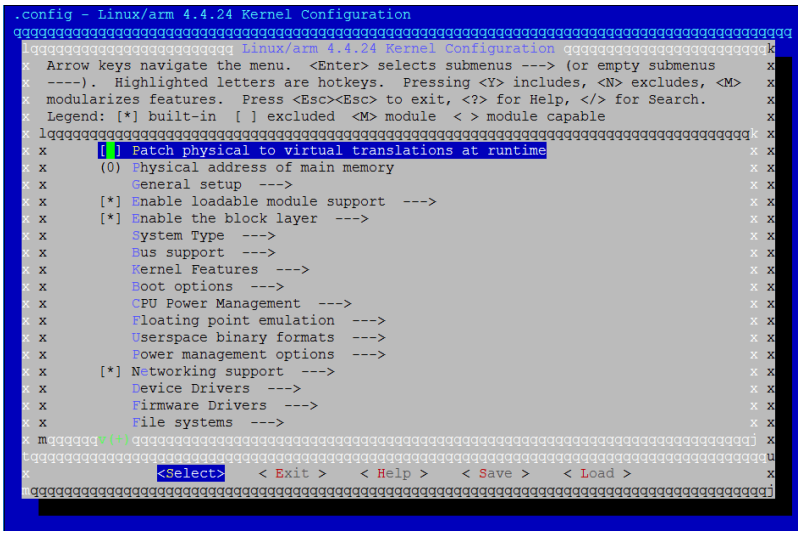
O próximo passo é clonar o repositório, *git://github.com/raspberrypi/linux.git* e, assim que a tarefa é finalizada, é necessário executar o comando *zcat /proc/config.gz > .config*. Esse comando gera um arquivo com as opções de configurações de núcleo atual. Tendo executado esses passos com sucesso, é possível retornar as configurações de espaço de arquivo *swap* para o padrão.

Na sequência, é necessário a instalação do pacote *ncurses-dev* para a realização da compilação do núcleo, o que permitirá entrar no modo de configuração de menus do núcleo. Utilizando o comando *make menuconfig*, uma janela semelhante à Figura 4.2 será aberta. Tendo acesso a configuração do núcleo, foram habilitados sete pacotes, mencionados anteriormente, que são essenciais para o funcionamento do LXC. Com essas opções habilitadas, foi possível recompilar o núcleo, sendo esse um processo complexo e que demorou cerca 3 horas para ser concluído na Raspberry Pi. Para isso, é necessário utilizar um

[7] Programa para acesso a SSH disponível para download em www.putty.org/.

conjunto de ferramentas[8] fornecidas pela Raspberry Pi que são responsáveis pela compilação do novo núcleo com as funcionalidades que foram habilitadas pelo *menuconfig*.

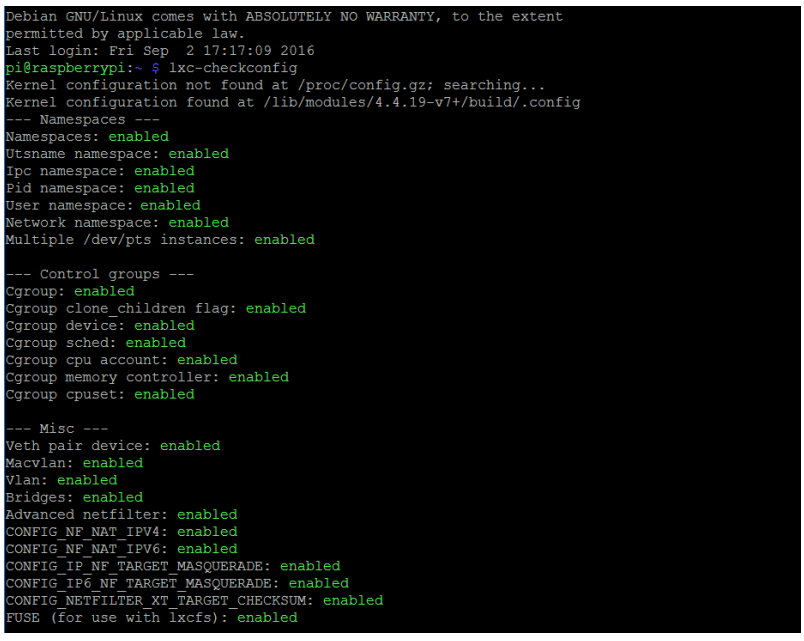
Figura 4.2 - Interface de configuração do núcleo



Fonte: Autor

Ao final desse processo, semelhante ao observado na Figura 4.1, é possível verificar na Figura 4.3 as funcionalidades que o núcleo suporta através do comando *lxc-checkconfig*.

Figura 4.3 – Funcionalidades do núcleo para executar LXC



Fonte: Autor

[8] Essas ferramentas são fornecidas no link git://github.com/raspberrypi/tools.git

Mesmo com os recursos habilitados, ainda foram necessários alguns passos adicionais para executar um Linux Contêiner. O primeiro deles é editar o arquivo */etc/fstab* e adicionar a linha *lxc /sys/fs/cgroup cgroup defaults 0 0*. O arquivo *fstab* é um arquivo de configuração que contém informações de todas as partições e dispositivos de armazenamento do computador. Assim, executando essa linha de comando, se está dizendo que o LXC será montado em */sys/fs/cgroup*, com tipo de sistema de arquivo *cgroup*, em modo *default* e sem verificação de *Dump* ou verificação *fck*. O *Dump* é um utilitário de *backup* e o *fck* é um utilitário de verificação do sistema de arquivos. O segundo passo é montar o arquivo após sua modificação utilizando o comando *mount*.

Para organizar os contêineres, foi criada uma pasta dentro de */var/lxc* nomeada por *guests*, onde todos os contêineres são armazenados. Cada contêiner fica literalmente em uma pasta separada, onde, com a ajuda da ferramenta *Debootstrap*, é possível transformar cada uma dessas pastas em um sistema de arquivo completo a partir de um diretório raiz e isolado dos demais. Isso acontece através da instalação de pacotes específicos da distribuição, como por exemplo, o *dpkg* (Debian). Ao terminar o processo de *Debootstrap*, é necessário utilizar o aplicativo *chroot* para modificar o sistema de arquivos do contêiner. Nesse ponto, é possível fazer modificações no arquivo de configuração do contêiner editando o arquivo *../container_teste/config*.

O LXC conta com uma série de comandos para o gerenciamento dos contêineres de um determinado hospedeiro. Por exemplo, *lxc-create*, *lxc-start*, *lxc-stop* e *lxc-destroy* são alguns dos comandos utilizados para criar, executar, parar e destruir um contêiner, respectivamente.

4.2 Gerenciador de nuvem

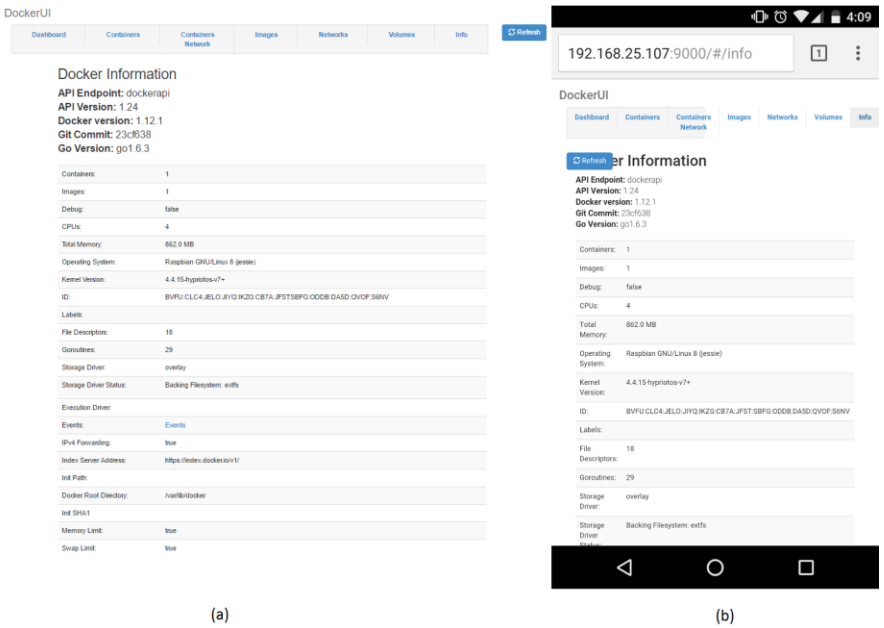
Apesar de ser possível a utilização de SSH, e controle por linhas de comandos para, o gerenciamento dos contêineres, optou-se por encontrar uma ferramenta que auxiliasse no gerenciamento através da utilização de uma interface gráfica. Tal objetivo é possível utilizando o DockerUI, que é um contêiner executado no hospedeiro e permite gerenciar outros contêineres. Esse projeto foi encontrado através do Hub Docker e foi desenvolvido e publicado pelo usuário Hypriot. A interface gráfica é acessível a partir de um browser. A Figura 4.6 (a) mostra as informações dos contêineres que estão na Raspberry Pi através de

acesso via *browser* em um computador pessoal; já a Figura 4.4(b) mostra as mesmas informações, mas acessando via *smartphone*.

Além de ser possível conferir informações dos contêineres, o DockerUI possibilita diversas outras funcionalidades. Na aba *Dashboard*, existe um gráfico onde se pode conferir o número de contêineres executando, parados ou fantasmas. Outra possibilidade é através da aba *Containers Network*, acompanhar um grafo com a rede formada pelos contêineres.

Sabendo um nome de repositório e o nome de imagem, é possível fazer requisições de imagens diretas da Internet e realizar o *download* para a Raspberry, ou dispositivo que esteja executando o DockerUI. Elas irão ficar armazenadas e acessíveis pela aba imagens, sendo possível deletar uma imagem a qualquer momento. Entre as opções dessa aba também está a criação de novos contêineres. Por último, a aba Contêineres, onde é possível inicializar, parar, reinicializar, pausar, destruir e remover contêineres, através de uma interface gráfica fácil de utilizar.

Figura 4.4 – Interface DockerUI



Fonte: Autor

4.4 Dificuldades encontradas

Uma dificuldade encontrada neste projeto foi o levantamento de requisitos necessários para que o *Linux Contêineres* funcionasse correto no hospedeiro. Embora haja grande suporte para o desenvolvimento de projetos em placas como a Raspberry Pi, a quantidade de

informação sobre trabalhos de virtualização é reduzida, sendo necessário adaptar ideias e soluções de projetos que não foram propriamente desenvolvidos para Raspberry.

Outra dificuldade encontrada, foi em relação a criar serviços que fossem possíveis implementar o conceito de elasticidade automática, assim alocando recursos de forma mais eficiente sobre a infraestrutura.

4.5 Considerações gerais

Neste capítulo foram explicados ajustes que foram empregados para a configuração do ambiente, assim, possibilitando que serviços de nuvem fossem utilizados em uma placa de baixo consumo.

No próximo capítulo serão apresentadas avaliações experimentais da arquitetura que foi desenvolvida neste projeto.

5 AVALIAÇÃO E TESTES

Para avaliar e validar a solução desenvolvida no Capítulo 4, este capítulo aborda os experimentos realizados e a discussão dos resultados obtidos. Inicialmente, é apresentada a plataforma experimental, para então discutir a metodologia utilizada na realização dos experimentos, os critérios adotados e cenários de teste. Por fim, conclui-se este capítulo com algumas considerações gerais.

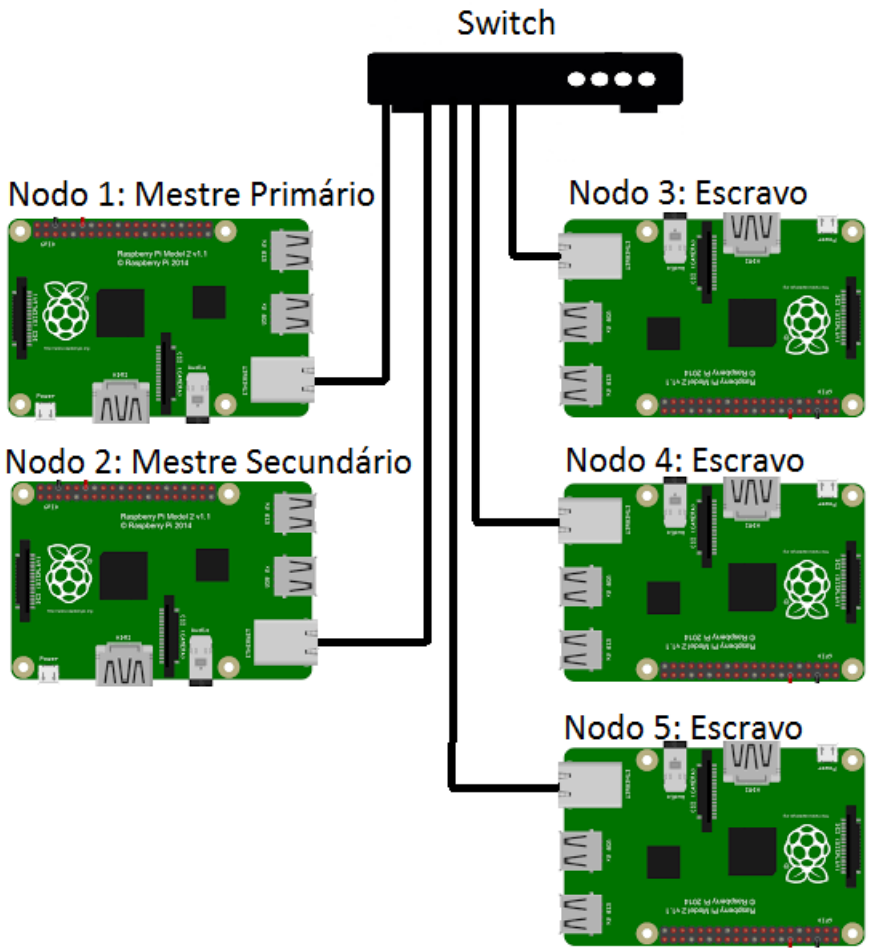
5.1 Plataforma de teste e metodologia

A plataforma experimental escolhida para realizar os experimentos é composta por uma rede formada por de um *cluster* com cinco placas Raspberry Pi 3, cada uma possuindo um processador Quad-Core ARM Cortex-A53, 1 GB de memória RAM e um cartão microSD de 32 GB como disco de armazenamento, e um notebook Dell Inspiron 15. Em cada uma das placas Raspberry é instalado a distribuição Raspbian modificada, que foi apresentada no capitulo 4, com suporte a LXC e Docker. A interligação das placas é feita através de um *switch* de 8 portas modelo TL-SG108 da empresa TP-Link. Para acesso ao *front-end*, é utilizado o notebook Dell Inspiron 15, que via *browser* acessa ao serviço de gerenciamento de infraestrutura que se encontra no nodo 1 da infraestrutura, Figura 5.1. O conjunto de placas é configurado possuindo um nodo mestre primário, nodo 1, e um nodo mestre secundário, nodo 2. Os demais nodos são escravos e apenas executam processos.

As medidas de consumo de recursos serão extraídas através dos comandos Linux *Free* e *Top*. Estes utilitários leem métricas do sistema de arquivos *proc* como, por exemplo, */proc/meminfo*, */proc/vmstat*, */proc/PID/smaps* [Kung, 2014]. As medidas de tempo para instanciação serão feitas a partir de um *script*, que busca o horário antes e depois da execução de uma instanciação através do comando *date*. Além disso, será feito a utilização do *NAS Parallel Benchmarks* (NPB) [NASA, 2014] para medir o desempenho de execução de tarefas no *cluster* e tempo de resposta a requisições utilizando a ferramenta Apache HTTP server *benchmarking*. Por fim, as medidas de disponibilidade serão extraídas através de *logs* e da ferramenta M/Monit[9].

[9] Ferramenta que auxilia no monitoramento de sistemas Unix, redes e serviços de nuvem. Disponível em https://mmonit.com/

Figura 5.1 – Arquitetura do Sistema



Fonte: Autor

5.2 Critérios de avaliação

Os critérios a serem utilizados para avaliar a Raspberry durante os testes propostos são:

- Consumo de recursos (memória e armazenamento) do sistema operacional e por diferentes contêineres.

- Tempo de instanciação de contêineres utilizando um *script* desenvolvido para capturar os tempos entre início e fim de uma instanciação.

- Desempenho de execução de tarefas no *cluster* utilizando *NAS Parallel Benchmarks* e *Apache HTTP server benchmarking tool*.

- Disponibilidade de serviço no *cluster*.

5.3 Cenários de teste

Serão analisados vários cenários de testes, descritos a seguir. Para cada um dos cenários, serão apresentados os resultados e conclusões parciais.

5.3.1 Cenário I: Recursos consumidos

Primeiramente, antes de saber a quantidade de recursos consumidos por LXC, foram realizadas medições com apenas o sistema operacional nativo em funcionamento. Para determinar quanto de memória (RAM) e espaço de armazenamento o sistema operacional utiliza, foi executado o comando *free –m*, que mostra informações relacionadas a memória em megabytes, como memória total, utilizada, livre, etc. Através desse comando foi observado que, para apenas manter o sistema operacional são necessários 50 MB dos 861 MB disponíveis, restando 811 MB livres para utilização de outros processos (Figura 5.2). Cabe ressaltar que os valores fornecidos na primeira linha são diferentes dos mencionados porque a memória reservada para *buffer* e *cache* (9 MB e 88 MB) são adicionados aos 50 MB já contabilizados pelo sistema operacional. Assim, o total de memória utilizada é cerca de 148 MB.

Figura 5.2 – Utilização de memória da Raspberry Pi 3

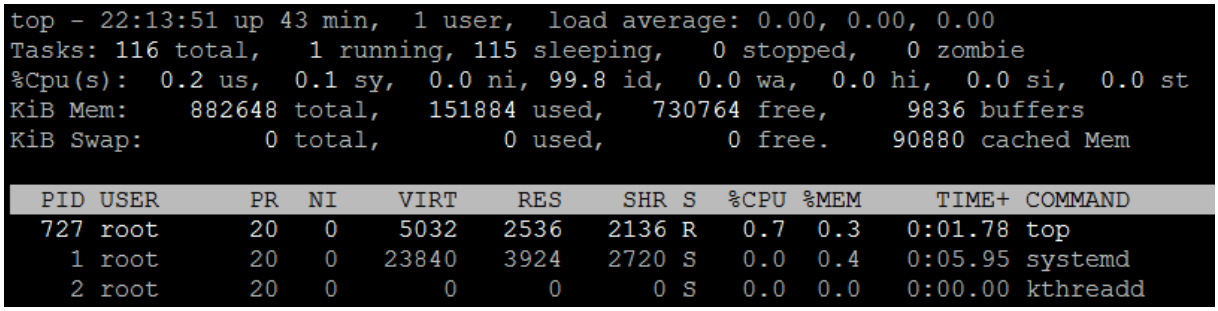
```
$ free -m
             total       used       free     shared    buffers     cached
Mem:           861        148        713          5          9         88
-/+ buffers/cache:         50        811
Swap:            0          0          0
```

Fonte: Autor

Além do comando *free –m*, durante as medições, foi utilizado o utilitário *Top*. Onde é possível verificar uma saída semelhante, Figura 5.3. Este utilitário traz consigo alguns detalhes adicionais em relação ao *free* como, por exemplo, total de processos em execução, utilização percentual do processador, utilização de memória, entre outros dados. Também foi verificado que o Raspbian consome 1 GB de espaço em disco dos 32 GB disponíveis utilizando o comando *df*.

`

Figura 5.3 – Utilização de memória e CPU da Raspberry Pi 3



Fonte: Autor

Sabendo que o total de memória disponível para utilizar com instâncias de contêineres é de 811 MB, foram realizadas algumas medições para saber os recursos que diferentes contêineres consomem. Por exemplo, ao instanciar um contêiner da imagem Rpi-dockerui, que é um contêiner responsável por disponibilizar uma interface de gerenciamento via *browser* ao usuário (Figura 4.6) foi verificado que esse contêiner utiliza 5,148 megabytes de memória RAM e cerca de 9,5 MB de espaço físico por instância.

Já um contêiner com um *servidor web* capaz de atender a requisições HTTP via *browser* e responder ao requisitante em qual contêiner e nodo respondeu a requisição, ocupa 27,13 MB. Nesse caso, o consumo de espaço em disco é de 118 megabytes. Esta aplicação *web* é composta por um arquivo *index.js*, que apresenta via *browser* o nome do contêiner e nodo que está atendendo a requisição. Além de um arquivo de dependências, responsável por construir e inicializar o *Node.js*[10]. Por fim, um arquivo *Dockerfile*, que parametriza qual imagem deve ser utilizada para construir a aplicação *web*, qual será a porta endereçada para o acesso externo, e inicialização dos arquivos listados anteriormente.

Outro exemplo é o Rpi-haproxy, sendo uma imagem para Raspberry de HAProxy, que é um *proxy* de alta disponibilidade, e é um *software open source* usado para melhorar o desempenho distribuindo as tarefas entre múltiplos servidores. Esse contêiner utiliza cerca de 16,81 megabytes e consome 181 megabytes de espaço em disco.

Apesar dos valores apresentados para contêineres acima, esse consumo pode variar muito, pois depende da aplicação e do propósito do contêiner. Na Tabela 2 é possível visualizar alguns contêineres e o consumo de memória utilizada. O Elastic Search é um poderoso mecanismo de pesquisa *open source*, que ajuda na análise e pesquisa de dados.

[10] Node.js é um ambiente de JavaScript Runtime desenvolvido para uma variedade de ferramentas e aplicações.

Enquanto isso, o Couchbase server[11], conhecido originalmente por Membase, é um banco de dados não relacional (NoSQL database) conhecido por sua escalabilidade e desempenho para aplicações iterativas. Esses componentes são utilizados em *Big Data*, mas não podem ser utilizados em uma mesma placa devido ao seu consumo de memória.

Tabela 2 – Contêiner vs Consumo de Memória

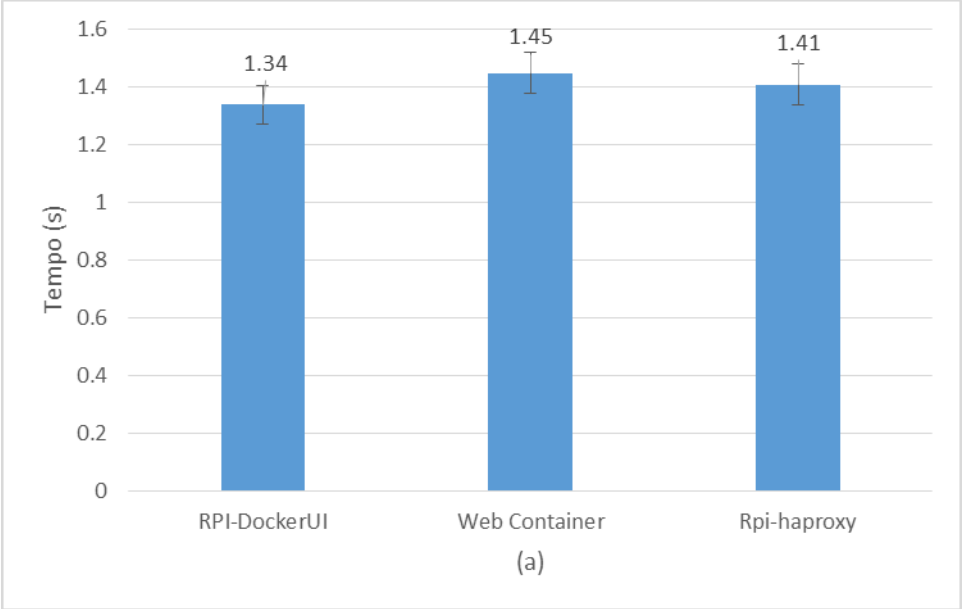
Contêiner	Consumo Memória
RPI-DockerUI	5,148 MB
Web Container	15,54 MB
Rpi-haproxy	21,54 MB
Elastic Search	202,2 MB
Couchbase Server	706,2 MB

Fonte: Autor

Adicionalmente, durante a análise de recursos consumidos foram conduzidos testes de tempo de instanciação por contêiner através de um *script*. O funcionamento desse *script* é realizado com uma sequência de passos. O primeiro deles é a quantidade de vezes que será instanciado determinado contêiner, seguido pela busca do horário antes e depois da execução da instanciação do contêiner. Em seguida, a instância criada é interrompida e destruída. Ao encerrar a execução dos passos anteriores, é calculado a média e o desvio padrão com base nos tempos coletados imediatamente antes e após a criação de cada contêiner.

[11] Desenvolvido pela Couchbase, Inc. Disponível em: www.couchbase.com/ e documentação sobre Couchbase em contêineres em: http://www.couchbase.com/containers

Figura 5.4 – Tempo de instanciação de contêiner(s)



Fonte: Autor

O processo descrito acima foi realizado para a instanciação dos contêineres RPI-DockerUI, *Web Container* e RPI-haproxy. No momento do teste existiam apenas processos do sistema operacional sendo executados. O teste foi realizado 30 vezes, onde a cada iteração foram coletados os valores da média das *n* vezes que foram instanciados o contêiner, Figura 5.4(a).

Comparando os valores de média e desvio padrão, é possível reparar que o tempo médio necessário para disponibilizar um contêiner RPI-DockerUI testado é em torno de 1,36 segundos e tem um desvio padrão entre a média das execuções de 0,035 segundos, representando uma variação de 2.8%. Assim, sendo possível afirmar que o tempo para a execução do processo de instanciação é consistente, ou seja, sem grandes variações no tempo total de uma instanciação. O mesmo foi observado para os contêineres *Web container* e Rpi-haproxy, no entanto, com um desvio padrão inferior ao contêiner RPI-DockerUI, entre a média das execuções, 24 e 21 milissegundos, respectivamente.

5.3.2 Cenário II: Desempenho utilizando *NAS Parallel Benchmarks*

Sabendo-se o comportamento dos contêineres referente ao seu consumo de recursos e tempo de instanciação, optou-se por realizar testes de maior complexidade. O primeiro deles foi a utilização de um conjunto de *benchmarks* que avaliasse o desempenho do *cluster* sobre o

ponto de vista do poder de processamento, modelo de comunicação e arquitetura. Para isso foi escolhido o *NAS Parallel Benchmarks* (NPB). O NPB é um conjunto de *benchmarks* para avaliação do desempenho de arquiteturas *multicores* desenvolvido pela Divisão de Supercomputação Avançada da NASA (*National Aeronautic and Space Administration*). Esses *benchmarks* são dedicados a computação e movimento de dados de aplicações de dinâmica dos fluidos computacionais em larga escala [BAILEY, 1991]. Apesar desse *benchmark* possuir 8 categorias de testes, o *cluster* desenvolvido neste trabalho será exposto a apenas 3 categorias selecionadas.

A primeira delas, *Block Tri-Diagonal Solver* (BT), consiste na resolução de um sistema sintético de equações diferenciais parciais não lineares, envolvendo dependências de dados globais.

O segundo, *Conjungate Gradient* (CG), utiliza o método da potência inversa para encontrar uma estimativa do menor autovalor de uma grande matriz esparsa, simétrica e positiva. Seu objetivo é acessar de forma irregular a memória e a comunicação.

Por último, o *Embarrassingly Parallel* (EP) gera pares de desvios aleatórios gaussianos utilizando o método polar de Marsaglia [MARSAGLIA, 1964], sendo uma aplicação computacional intensiva.

Os testes BT e CG foram utilizados para testar a infraestrutura perante a acesso a grande quantidade de dados, com padrões irregulares e regulares. Por outro lado, o teste EP foi conduzido para verificar como essa infraestrutura se comporta perante a uma aplicação de CPU intensiva.

Além da divisão por categorias, o NPB possui diferentes classes para auxiliar na execução de testes em diversas arquiteturas e hardwares com capacidades de cálculo e comunicação variáveis. Para arquiteturas simples, usam-se classes menores, como a S e W. Para arquiteturas intermediárias existem as classes A, B e C, sendo que de uma classe para a próxima o tamanho aumenta em 4 vezes. Já as classes D, E e F, são destinadas a arquiteturas de grande porte e a necessidade computacional aumenta dezesseis vezes de uma classe para a próxima.

Os testes do *benchmark* foram gerados utilizando a implementação OpenMPI, com o compilador mpif77 (OpenMPI) e sem nenhum parâmetro adicional. O *benchmark* EP foi gerado para a classe C, já os outros dois testes foram gerados para a classe B. Cada classe de teste foi executada 10 vezes. O tempo decorrido em segundos para execução é expresso como a média aritmética dos tempos de cada execução, juntamente com o intervalo de confiança (C_i) de 95%.

Algumas das classes executam apenas com um número definido de *threads*, normalmente potências de dois, como o *benchmark* CG e EP, outros utilizam raízes quadradas perfeitas, como o *benchmark* BT. Assim, o *cluster* foi submetido a cargas com um número de *threads* potência de dois até o número de *threads* ultrapassarem o número de núcleos do *cluster*. Foram executadas cargas com 1, 4, 8, 16, 32 *threads*. Com exceção da carga BT, que suporta apenas raízes quadradas perfeitas como número de *threads* e por consequência, foram usadas 1, 4 e 16 *threads*.

Analisando a Tabela 3, é possível observar que o *cluster* obtém melhor desempenho ao possuir todos seus *cores* ocupados, o que era esperado em função da Raspberry Pi possuir um processador quad-core ARM Cortex-A53 e processadores com arquitetura *multicore* oferecem um aumento de desempenho, principalmente com aplicações que utilizam paralelismo [GEPNER. P 2006].

Tabela 3 – Tempos e intervalos de confiança

Benchmark-Classe	*Threads*	*Cluster*	
		Tempo (s)	C_i
CG-B	1	3199.65	12.58
	4	857.5062	6.61
	8	565.9541	8.53
	16	469.7419	6.18
	32	718.7051	7.46
EP-C	1	2655.53	4.05
	4	677.75	0.31
	8	340.343	0.14
	16	171.2606	0.19
	32	167.9381	0.28
BT-B	1	5829.53	4.47
	4	1589.9	1.49
	16	570.32	2.58

Fonte: Autor

Analisando os resultados dos testes CG e BT (submetem a arquitetura a uma tarefa que exige acesso a grande quantidade de dados, com padrões irregulares e regulares). O fato de a Raspberry possuir uma cache L1 de 32 kB e L2 com 512 kB podem inviabilizar a utilização para aplicações que tenham muitos acessos à memória.

Por outro lado, os resultados obtidos no teste EP, que é uma aplicação de CPU intensiva, são promissores. É possível observar o decréscimo praticamente linear conforme

mais *threads* são adicionadas para execução da aplicação, até a saturação do *cluster*, onde o número de threads ultrapassa a quantidade de cores disponíveis na arquitetura.

5.3.3 Cenário III: Desempenho utilizando Docker Swarm

Uma segunda abordagem, para testar o desempenho do *cluster*, foi submeter à infraestrutura a acessos via *Hypertext Transfer Protocol* (HTTP) e analisar seu desempenho para atender um volume *n* de requisições, onde *n* é um valor maior que zero. As requisições são disparadas através de uma ferramenta capaz de simular requisições advindas de *web browsers*. Essas requisições são recebidas pelo nodo 1, responsável por fazer o balanceamento de carga e distribuir entre os demais nodos. Após uma requisição ser processada, ela é devolvida ao nodo responsável pelo balanceamento de carga, que por sua vez, encaminha as respostas aos clientes.

Para executar o experimento foi configurado um contêiner RPI-haproxy no nodo 1, que serve de balanceador, responsável por distribuir requisições no *cluster*. Além disso, utilizando o modo *Docker Swarm*, introduzido na seção 3.1, foi criado um serviço com base no contêiner *servidor web*, que foi descrito na seção 5.3.1, e que possuem conteúdo idêntico entre si. Ao longo do experimento foi variada a quantidade de nodos trabalhadores e foram extraídos dados sobre o tempo de resposta. Sendo um dos objetivos verificar como a elasticidade, mesmo que manual, influência no desempenho.

Para realizar o teste foi utilizado o *Apache HTTP server benchmarking tool*, httpd.apache.org. Essa ferramenta foi desenvolvida para quantificar o desempenho da instalação de um *HTTP server*. Utilizando esse *benchmark* foi possível submeter à arquitetura a uma quantidade de requisições com um determinado grau de concorrência de usuários.

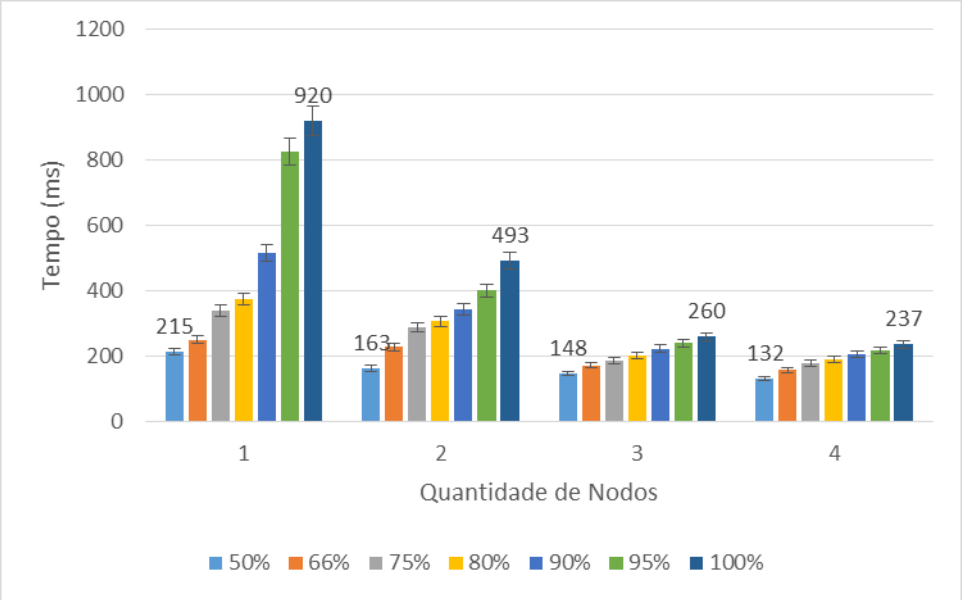
Durante todo o teste, o nodo 1, responsável pela distribuição de requisições, foi mantido *online*. Já os demais nodos foram ligados um a um. Sendo assim, inicialmente apenas o nodo 1 e 2 estavam trabalhando para atender a demanda do teste. Após a conclusão da tarefa, outro nodo foi adicionado aos nodos que já estavam respondendo requisições. A mesma abordagem foi adotada todos os nodos estarem aptos a processar uma tarefa. Em todos os casos, foi utilizado uma quantidade total de 10000 requisições direcionadas ao nodo 1.

Esse teste foi executado até obter intervalo de confiança superior a 95% e os tempos em milissegundos apresentados na Figura 5.5 são a média aritmética das *n* execuções do teste para cada arranjo da arquitetura. A Figura apresenta a quantidade de requisições que são atendidas em um determinado intervalo de tempo em milissegundos. Por exemplo, a primeira

coluna dos valores obtidos para apenas um nodo trabalhando representa que 50% das requisições foram atendidas com menos de 215 milissegundos. Já a última coluna para apenas um nodo indica que dentro do universo das 10000 requisições, existem requisições que demoraram até 920 milissegundos para serem atendidas.

Assim, é possível observar que conforme novos nodos são adicionados o tempo de resposta para as requisições vai reduzindo a uma taxa considerável para os primeiros dois nodos adicionados. No entanto, com um quarto nodo é possível observar que não ocorreu uma grande melhora no tempo para atender as requisições. Sendo talvez um indicio que a adição de mais nodos não melhorariam o desempenho para esse teste.

Figura 5.5 – Tempo de resposta a 10.000 requisições HTTP



Fonte: Autor

5.3.4 Cenário IV: Disponibilidade utilizando *Docker Swarm*

Por fim, como qualquer outra infraestrutura, existe a possibilidade de indisponibilidade parcial ou total do serviço oferecido. Sendo assim, a infraestrutura foi submetida a testes de indisponibilidade parcial de componentes. O primeiro objetivo deste experimento foi averiguar o que acontece quando um nodo aleatório fica indisponível. No segundo teste, um dos nodos foi submetido a *reboot*. Para realização deste teste foi estabelecido o mesmo serviço *web* utilizado no cenário 3. Durante o experimento, esse serviço sofreu variação de 10, 15 e 20 contêineres distribuídos pela infraestrutura de *hardware*.

Inicialmente, o nodo mestre do *cluster* distribuiu de forma igualitária os contêineres entre os nodos. Quando o sistema estabiliza, ou seja, tem todos os contêineres sendo executados, a interface ethernet de uma das placas é desligada. A primeira fase do teste é concluída quando o sistema distribui os contêineres executados pelo nodo que sofreu a indisponibilidade, entre os demais nodos. Com isso, possuindo a mesma quantidade de contêineres executados do início do experimento. Na segunda fase do experimento a interface ethernet é reativada. O final do experimento é quando o nodo que sofre indisponibilidade volta a funcionar, e a infraestrutura tem sua carga de inicial restaurada.

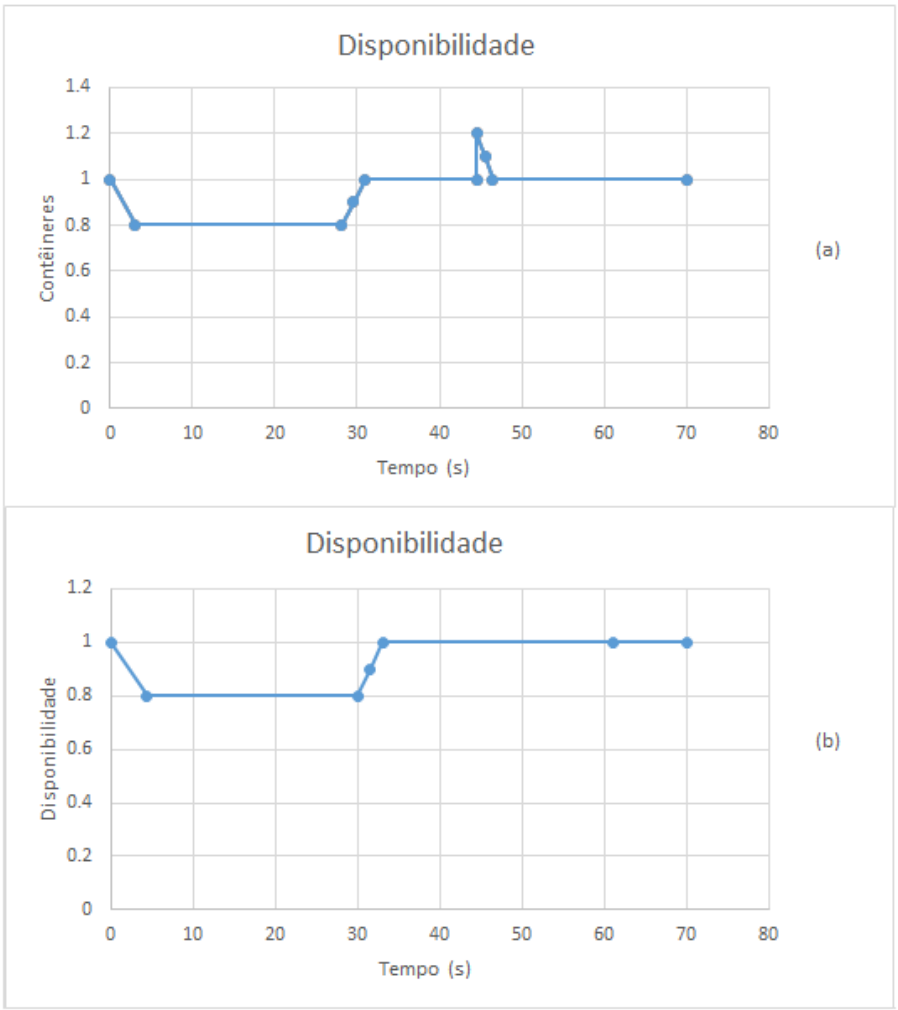
O segundo teste de disponibilidade de infraestrutura é realizado através de *reboot* aleatório de um nodo. Assim como no primeiro teste, a primeira fase do teste é concluída quando o sistema se estabiliza distribuindo o número de contêineres inicial entre os demais nodos, e, a segunda, quando o nodo que sofreu *reboot* volta a funcionar e tem sua carga de contêineres reestabelecida.

Experimentalmente, através de observações e medições, é possível verificar que o tempo requerido para o sistema estabilizar para a primeira fase e a segunda fase do teste na Figura 5.6, no caso onde existem 10 contêineres distribuídos entre os nodos. A disponibilidade ao longo do tempo, Figura 5.6(a), é referente ao teste onde a infraestrutura se recupera de um nodo tendo sua interface ethernet desconectada. Já a Figura 5.6(b) mostra o resultado de uma recuperação devido ao *reboot* de um nodo. Ao comparar as linhas do tempo é possível verificar que o *cluster* com modo *swarm* ativo, detecta em aproximadamente 3 segundos que um nodo não está respondendo. No entanto, apenas aos 30 segundos começa a estabilizar a oferta de contêineres na infraestrutura, que é o período de tempo que o mecanismo demora para se comunicar com todos os nodos, e balancear novamente as cargas do sistema.

No teste 1 o período necessário para a infraestrutura se recuperar da indisponibilidade do nodo. No caso 5.6(a), quando a infraestrutura volta a oferecer a mesma quantidade de contêineres, a interface ethernet do nodo é reativada. A partir desse momento, é necessário cerca de 12,5 segundos para ele voltar a operar. Quando o nodo indisponível retorna a ser parte do *cluster*, ele acaba gerando uma carga de 12 contêineres. Isso se deve ao fato de os contêineres continuarem com *status* executando, mesmo estando indisponíveis, e com isso gerando uma oferta maior que a desejada. Logo que o *swarm* verifica automaticamente essa super oferta, ele acaba rebalanceando as cargas nos nodos, e com isso oferecer apenas 10 contêineres.

Já no caso 5.6(b), é possível verificar comportamento semelhante ao teste 1, no entanto, a grande diferença está no momento onde o nodo indisponível retorna para o *cluster*, isso ocorre aproximadamente aos 61 segundos, onde o nodo recuperar-se do *reboot* e volta a possuir carga do rebalaceamento. Contudo nesse caso não se observa um pico aos 61 segundos, representado por um ponto, isso ocorre devido à quando o nodo é reinicializado seus contêineres ficam com *status* parado e com isso quando o nodo retorna, não é possível observar pico de maior oferta de contêineres.

Figura 5.6 – Tempo de indisponibilidade parcial com 10 contêineres



Fonte: Autor

Após isso, foi gerado o mesmo experimento com 15 e 20 contêineres disponíveis no *cluster*. O comportamento da infraestrutura com um número maior de contêineres disponíveis foi semelhante ao observado no experimento anterior. Com a exceção de que, devido ao

número de contêineres por nodo, resultou-se em um intervalo de tempo maior na fase de instanciação de contêineres. O experimento foi realizado 10 vezes, sendo o tempo de execução em segundos expresso como a média aritmética dos tempos de cada execução e o intervalo de confiança (C_i) de 95%.

Tabela 4 – Tempo de indisponibilidade parcial teste 1

Quant. Contêineres	Experimento	Primeiro Fase		Segunda Fase	
		Tempo (s)	C_i	Tempo (s)	C_i
10	Ethernet	29.94	0.87	45.49	0.56
15	Ethernet	31.47	0.68	47.08	0.74
20	Ethernet	33.13	0.74	48.47	0.68

Fonte: Autor

Tabela 5 – Tempo de indisponibilidade parcial teste 2

Quant. Contêineres	Experimento	Primeiro Fase		Segunda Fase	
		Tempo (s)	C_i	Tempo (s)	C_i
10	Reboot	33.14	1.67	61.1	1.43
15	Reboot	34.56	1.43	62.48	2.17
20	Reboot	36.17	1.80	64.03	1.92

Fonte: Autor

É possível observar através dos dados apresentados na Tabela 4 e 5, que conforme o número de contêineres aumenta, o tempo necessário para a infraestrutura se recuperar aumenta na casa de aproximadamente 1.5 segundos, sendo que a diferença entre tempos é relacionada ao tempo para alocar contêineres adicionais. Também é possível observar que a variação é maior nos testes onde um nodo sofre *reboot*, possivelmente, pelo fato do tempo necessário para a inicialização do sistema operacional.

5.4 Considerações finais

Posto todos os cenários de testes, foi verificado com os resultados obtidos, que é possível usar um *cluster* com placas Raspberry Pi 3 para prover infraestrutura de hardware para computação em nuvem. Apesar da Raspberry possuir recursos limitados se comparado com outras infraestruturas, ela possui poder computacional para executar diversas tarefas em um tempo razoável. Principalmente, quando essas tarefas são executadas por um *cluster* de placas Raspberry.

Outro ponto muito importante é a utilização do LXC e Docker, que possibilitam o uso inteligente dos recursos limitados da placa. Principalmente com o modo *swarm*, que faz com

que várias placas tenham um comportamento de troca de informações e colaborem entre si para realizar tarefas.

6 CONCLUSÃO

Nos últimos anos, o significativo aumento da computação em nuvem tem transformado a rotina de nossa sociedade. É praticamente impossível encontrar alguém que não utilize serviços como e-mail, redes sociais, e que realize diversas atividades que antigamente eram feitas de forma presencial pela Internet. Contudo, para manter todos esses serviços funcionando é necessário uma infraestrutura que mantenha a disponibilidade e qualidade de serviço. Assim, surgem cada vez mais *data centers* para manter esses serviços funcionando. Em 2014, foi estimado que 1.8% de todo o consumo energético dos Estados Unidos [SHEHABI, 2016] devesse a energia elétrica gasta por *data centers*, resultando em problemas como consumo de energia para alimentar essas infraestruturas e a poluição, causada pela necessidade do aumento na capacidade de produção de energia elétrica.

Em virtude disso, atualmente, busca-se métodos e práticas para utilizar a energia elétrica de forma eficiente e com isso minimizar os efeitos no ambiente. Com isso, Computação Verde tem sido um assunto largamente discutido. Essa abordagem procura melhorar a eficiência energética de componentes eletrônicos, incluindo grandes sistemas como *data centers*. Assim, diversas técnicas são propostas em nível de hardware (processador, memória, rede, etc.) e software (funcionalidades no sistema operacional e técnicas em nível de aplicações), sendo a virtualização outra ótima alternativa para aumentar a taxa de recursos utilizados em infraestruturas de hardware.

Com base nesses conceitos, ao longo deste trabalho foram abordados conceitos e tecnologias de virtualização, e de computação em nuvem. Com base nesse estudo, foi proposto, implementado e avaliado uma infraestrutura de placas de baixo consumo para oferecer serviços em nuvem.

Assim, usando um *cluster* com 5 placas Raspberry Pi 3, que são placas de baixo consumo de energia elétrica, foram realizados testes de consumo de memória, desempenho e disponibilidade, a fim de verificar a possibilidade de usar infraestruturas construídas com esse tipo de hardware como suporte a serviços em nuvem.

Durante o estudo para realização deste trabalho, verificou-se a necessidade de utilizar um tipo de virtualização com um baixo consumo de recursos em virtude de restrições de recursos de memória da placa Raspberry. A alternativa encontrada foi à utilização de LXC, uma forma de virtualização recente que é realizada em nível de software.

Os testes realizados sobre a infraestrutura desenvolvida, como prova de conceito, neste trabalho, mostraram que tal infraestrutura é capaz de executar diversas tarefas e aplicações.

Principalmente as aplicações que tem característica de serem CPU intensivas tem grande potencial de serem executadas com bom desempenho em uma infraestrutura formada por placas de baixo consumo. No entanto, devido ao fato de possuir processadores com baixa capacidade de cache, e não possuir conexão Gigabit Ethernet, o *cluster* com placas Raspberry Pi não é a melhor escolha para aplicações que demandem muito acesso a memória e elevada comunicação entre nodos.

A contribuição do trabalho vem do estudo da possibilidade de utilizar uma forma de virtualização em nível de sistema operacional, e que vem sendo utilizada por grandes empresas do mercado de tecnologia, parecendo ser mais eficiente que formas “tradicionais” de virtualização. Principalmente para máquinas que não disponibilizam de uma quantidade abundante de recursos.

Por fim, como sugestão para trabalhos futuros, seria interessante realizar uma análise de consumo energético e como utilizando “*time-to-solution*” [PADOIN, 2016] poderia melhorar a eficiência da infraestrutura desenvolvida para este projeto. Outra sugestão seria o emprego de elasticidade automática, alocar e desalocar contêineres conforme a demanda momentânea que a infraestrutura está atendendo. Também seria interessante utilizar área de *swap* para ver o quanto ajudaria no desempenho da infraestrutura. Uma última ideia seria a utilização de computação hibrida, já que a Raspberry possui uma GPU, e assim, obter um melhor desempenho da infraestrutura.

REFERÊNCIAS

BAILEY, D. H., BARSZCZ, E., BARTON, J. T., BROWNING, D. S., CARTER, R. L., DABUM, L., FATOOHI, R. A., FREDERICKSON, P. O., LASINSKI, T. A., SCHEIBER, R. S., SIMON, H. D., VENKATAKRISCHNAN, V., and WEERATUNGA, S. K.. **The NAS Parallel Benchmarks - Summary and Preliminary Results**. In Proceedings of the 1991 ACM/IEEE Conference on Supercomputing, Supercomputing '91, p. 158–165, New York, NY, USA. ACM

BODEN RUSSELL, **Passive Benchmarking with Docker LXC, KVM & OpenStack**, <http://www.slideshare.net/BodenRussell/kvm-and-docker-lxc-benchmarking-with-openstack>, 2014. Acessado em: 29 setembro 2016.

CARISSIMI, A. **Virtualização: Princípios Básicos e Aplicações**, UFRGS, 2009.

CloudStack, **CloudStack Official Page** <https://cloudstack.apache.org/>, 2016. Acesso em: 15 maio 2016.

COCOZZA, F., LOPEZ, G., MARIN, G., VILLALON, R., and ARROYO, F. **Cloud Management Platform Selection: A Case Study in a University Setting**. CLOUD COMPUTING 2015, [S.l.], 2015.

FENG W., "**Making the Case for Efficient Supercomputing**," ACM Queue, vol. 1, no. 7, p. 54–64, 2003.

Gartner Source **Modernization and Digital Transformation Projects Are Behind Growth in Enterprise Application Software Market**. <http://www.gartner.com/newsroom/id/3119717>, 2015. Acesso em: 21 maio 2016.

Gartner Source **Worldwide Public Cloud Services Market Is Forecast to Reach $204 Billion in 2016**. <http://www.gartner.com/newsroom/id/3188817>, 2015. Acesso em: 20 maio 2016.

GEPNER, P. and KOWALIK, M. F. **Multi-Core Processors: New Way to Achieve High System Performance**. International Symposium on Parallel Computing in Electrical Engineering. [S.l.] set, 2006.

GOLDBERG, R.P. **Architecture of Virtual Machines**. In Proceedings of Workshop on Virtual Computer Systems. Cambridge, MA, USA, 1973. pp 74-112.

IBM **Power Instruction Set Architecture**. [S.l.] v 2.04. IBM Corporation.

IBM Corporation. **IBM Systems Virtualization Version 2 release 1** (2005). [S.l.] Disponível em <http://public.boulder,ibm.com/infocenter/eicay.pdf>. Acesso em: 24 maio 2016.

LAUREANO, M.; MAZIERO, C. A.. Virtualização: Conceitos e Aplicações em Segurança. **Minicursos em VIII Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais**. Porto Alegre: Sbc, 2008. p. 139-187.

LOSCHWITZ, Martin Pacote completo. **Linux Maganize**, [S.l.], v. 113, p. 31-35, out. 2014.

KUNF, F. **Memory Inside Linux Containers**, [S.l.], disponível em <https://fabiokung.com/2014/03/13/memory-inside-linux-containers/>. Acesso em: 3, out, 2016.

MARSAGLIA, G. and BRAY, T. A., **A Convenient Method for Generating Normal Variables**, Vol. 6, No. 3, p. 260-264, 1964.

MARON, C. A. F., GRIEBLER, D., VOGEL, A., and SCHEPKE, C. **Avaliação e Comparação do Desempenho das Ferramentas OpenStack e OpenNebula**. 2014. In 12th Escola Regional de Redes de Computadores (ERRC), Canoas: Sbc, 2014. p. 1–5.

MELL, P. GRANCE, T. The NIST Definition of Cloud Computing. **National Institute of Standards and Technology Special Publication**. [S.l.], n 800-145, 2011.

OpenNebula (2016), OpenNebula Official Page (https://opennebula.org/). Acesso em maio 2016.

OPENSTACK (2016), OpenStack Official Page (https://www.openstack.org/). Acesso em maio 2016

PADOIN, Edson Luiz, **LinkEnergy-aware load balancing approaches to improve energy efficiency on HPC systems**, Tese de doutorado. Universidade Federal do Rio Grande do Sul. Instituto de Informática. Programa de Pós-Graduação em Computação, Porto Alegre, BR-RS, 2016

POPEK, G. and GOLDBERG, R. **Formal requirements for virtualizable third Generation architectures**. Communications of the ACM, 1974, 17(7):412–421.

ROSENBLUM, M. **The reincarnation of virtual machines**. Queue Focus – ACM Press, 2004. p. 34-40.

SHEHABI, A., SMITH, S., SARTOR, D., BROWN R., HERRLIN M., KOOMEY, J. MASANET E., **UNITED STATES DATA CENTER ENERGY USAGE REPORT**, Enerst Orlando Lawrence Berkeley National Laboratory, LNBL-1005775, June, 2016.

SHROFF, G. **Enterprise Cloud Computing: Technology, Architecture, Applications**. Cambridge University Press, 2010.

SOUSA, F. R. C.; MOREIRA, L. O.; MACHADO, J. C. **Computação em Nuvem: Conceitos, Tecnologias, Aplicações e Desafios**. Fortaleza, 2009.

STRACHEY, C. **Time sharing in large, fast computers**. IFIP Congress, 1959.

TAURION, C. **Cloud Computing: computação em nuvem: transformando o mundo da tecnologia da informação**, Editora Brasport: Rio de Janeiro, Brasil, 2009.

THEMAGPI, Inside Raspberry PI 3, **The Magpi – The official Raspberry Pi magazine**, [S.l], v 43, p. 8, março 2016.

UHLIG, R., NEIGER, G., RODGERS, D., SANTONI, A., ANDERSON, F. M. A., BENNETT, S., KAGI, A., LEUNG, F., and SMITH, L. **Intel virtualization technology**. IEEE Computer, 2005.

UNIX Time-Sharing System: Unix Programmer's Manual, Seventh Edition, V. 1, January 1979.

STRACHEY, C. "**Time sharing in large, fast computers**" in Proceedings of the IFIP Congress, [S.l.] pp.336-341, 1959

VECCHIOLA, C., BUYYA, R. and PANDLEY, S. **Cloud Computing**, 2009.

YEN, C. H. **Solaris operating system - hardware virtualization product architecture**. Technical Report 820-3703-10, Sun Microsystems, 2007.