CENTRO PAULA SOUZA

FACULDADE DE TECNOLOGIA DA ZONA LESTE

ARCANJO, THAIS

BARCELOS, STEPHANIE

PORTILHO, ESDRAS

SiGAW: SISTEMA GERENCIADOR DE APLICAÇÕES WEB

São Paulo

2015

CENTRO PAULA SOUZA

FACULDADE DE TECNOLOGIA DA ZONA LESTE

ARCANJO, THAIS

BARCELOS, STEPHANIE

PORTILHO, ESDRAS

SiGAW: SISTEMA GERENCIADOR DE APLICAÇÕES WEB

Trabalho de Conclusão de Curso apresentado à Faculdade de Tecnologia da Zona Leste no curso de Análise e Desenvolvimento de Sistemas, como exigência para obter o título de Tecnólogo em Análise e Desenvolvimento de Sistemas.

São Paulo

2015

ARCANJO, THAIS

BARCELOS, STEPHANIE

PORTILHO, ESDRAS

SiGAW: SISTEMA GERENCIADOR DE APLICAÇÕES WEB

Esdras Portilho, Stephanie Barcelos e Thais Arcanjo – Faculdade de Tecnologia da Zona Leste, São Paulo, 2015

48p.

Orientador: Welligton Pinto de Oliveira

Trabalho de Conclusão de Curso – Faculdade de Tecnologia de Zona Leste

CENTRO PAULA SOUZA

FACULDADE DE TECNOLOGIA DA ZONA LESTE

ARCANJO, THAIS

BARCELOS, STEPHANIE

PORTILHO, ESDRAS

SiGAW: SISTEMA GERENCIADOR DE APLICAÇÕES WEB

Trabalho de Conclusão de Curso apresentado à Faculdade de Tecnologia da Zona Leste no curso de Análise e Desenvolvimento de Sistemas, como exigência para obter o título de Tecnólogo em Análise e Desenvolvimento de Sistemas.

Aprovado em:

Banca Examinadora

Orientador: Prof. Welligton Pinto de Oliveira
Instituição: Fatec-ZL

Prof. Dr. ou Me. _____ Instituição: _____

Julgamento: _____ Assinatura: _____

Prof. Dr. ou Me. _____ Instituição: _____

Julgamento: _____ Assinatura: _____

Prof. Dr. ou Me. _____ Instituição: _____

Julgamento: _____ Assinatura: _____

São Paulo, ___ de _____ de 20___.

Dedicamos este trabalho de conclusão de curso aos nossos familiares, amigos e professores e coordenadores do curso que de muitas formas incentivaram e ajudaram para que fosse possível a concretização deste trabalho.

AGRADECIMENTOS

Agradecemos a todos que contribuíram no decorrer desta jornada, em especialmente: A Deus, a quem devemos a nossas vidas. A nossas famílias que sempre nos apoiou nos estudos e nas escolhas tomadas. Ao orientador Prof. Wellington Pinto de Oliveira que teve papel fundamental na elaboração deste trabalho. E aos nossos colegas pelo companheirismo e disponibilidade para nos auxiliar em vários momento

"Não é o mais forte que sobrevive, nem o mais inteligente, mas o que melhor se adapta às mudanças"
(Charles Darwin).

RESUMO

ARCANJO, Thais e BARCELOS, Stephanie e PORTILHO, Esdras. **SiGAW:** Sistema Gerenciador de Aplicações WEB, 48p. Trabalho de conclusão de curso, Faculdade de Tecnologia da Zona Leste, São Paulo, 2015.

Com a globalização, seguido da necessidade de práticas mais sustentáveis, à recentes pressões econômicas além dos frequentes desastres naturais resultaram em grandes exigências com relação à disponibilidade, dimensionamento e eficiência quando se fala em TI. Para ajudar a essas novas mudanças muitos passaram a desenvolver sistemas através do novo paradigma de TI, o cloud computing, também conhecido no Brasil como computação em nuvem ou computação nas nuvens, basicamente esse conceito tende a deslocar a localização de toda a infraestrutura computacional para a rede. Com base nas tecnologias estudadas e apresentadas nesse projeto desenvolvemos o SiGAW, um Sistema de Gerenciamento de Aplicações Web, que busca virtualizar aplicações em nuvem através da linguagem JavaScript que possibilitará o usuário rodar aplicações através de containers por Docker exultando em uma aplicação executada no browser, que levará para qualquer usuário um gerenciador de aplicações na nuvem, podendo continuar sua utilização de qualquer lugar, buscando diminuir consideravelmente os custos de software e principalmente de hardware para os usuários. Como metodologia de pesquisa, foi feito um levantamento de referenciais teóricos com livros, artigos e dissertações no intuito de implementar o que está sendo proposto através do sistema SiGAW.

Palavras Chave: Cloud Computing, Computação em Nuvem, SiGAW, Web, Virtualizar, JavaScript, Docker.

ABSTRACT

ARCANJO, Thais and BARCELOS, Stephanie and PORTILHO, Esdras. **SiGAW:** Sistema Gerenciador de Aplicações WEB, 48p. Final Project, Faculdade de Tecnologia da Zona Leste, São Paulo, Brazil, 2015.

With globalization, followed by the need for more sustainable practices, recent economic pressures and even the natural disasters lead to high demands regarding the availability, scalability and efficiency when it comes to IT.

To help these new changes many started to develop systems through the new paradigm of IT, cloud computing, also known in Brazil as “computação em nuvem” or “computação nas nuvens”, basically this concept tends to shift the location of the entire computing infrastructure to network.

Based on the studied technologies and presented this project we developed the SiGAW, a Web Application Management System, which seeks to virtualize cloud applications through JavaScript language that will allow the user to run applications through containers in Docker exulting in an application running in the browser, that will provide to any user an application manager in the cloud that will also offer the possibility to continue its use in any place, seeking to considerably reduce the costs of software and especially hardware for users.

As a research methodology, it was made a survey of theoretical references with books, articles and dissertations in order to implement what is being proposed by SiGAW system.

LISTA DE SIGLAS

CPU	Central Processing Unit
E/S	Entrada e Saida
GB	GigaByte
HD	Hard Disk
HTML	Linguagem de Marcação de Hipertexto
HTTP	Protocolo de Transferencia Hipertexto
HTTPS	Protocolo de Transferencia Hipertexto Seguro
IP	Internet Protocol
LXC	Linux Container
MVC	Model View Control
OS	Operacional System
PVR	Personal Video Recorder
RAM	Random Access Memory
SiGAW	Sistema Gerenciador de Aplicações Web
SSL	Secure Socket Layer
TCP	Protocolo de Transmissao
UTF	Unicode Transformation Format
VM	Virtual Machine

LISTA DE FIGURAS

Sumário

1 Introdução

A evolução das tecnologias da informação vem promovendo diversas mudanças na sociedade em geral. Entre elas está a disponibilização de uma quantidade cada vez mais crescente de informações, resultado principalmente do aumento da capacidade de processamento e armazenamento.

Este fenômeno torna-se cada vez mais evidente e vem sendo observado por diversos estudiosos da área. Em 2003 o mundo produzia entre um e dois exabytes de informação nova por ano, ou seja, algo em torno de 250 megabytes para cada habitante na Terra (LYMAN; VARIAN, 2003). Um exabyte equivale a pouco mais de um bilhão de gigabytes. Estima-se que documentos impressos, que eram o meio mais comum de informação textual há algumas décadas, hoje representem apenas 0,003% da informação gerada anualmente (LYMAN; VARIAN, 2003).

O termo cloud computing também conhecido no Brasil como computação em nuvem ou computação nas nuvens está associado a um novo paradigma na área de computação. Basicamente, esse conceito tende a deslocar a localização de toda a infra-estrutura computacional para a rede. Com isso, os custos de software e principalmente de hardware podem ser consideravelmente reduzidos (TAURION, 2009).

Este assunto esteja sendo amplamente discutido nos dias de hoje, ainda não há uma definição completa do termo. Na literatura, pode-se encontrar muitas indefinições que em algumas vezes podem ser semelhantes, e em outras podem apresentar conceitos diferentes. Por exemplo, alguns defendem que a escalabilidade e o uso otimizado dos recursos são características chave em cloud computing, enquanto outros discordam, afirmando que esses elementos não são características, e sim requerimentos de uma infra-estrutura que suporta esse novo paradigma da computação (TAURION, 2009).

Desse modo, a utilização da computação em nuvem é uma solução plausível que possibilitará o usuário rodar aplicações através de containers por Docker

exultando em uma aplicação executada no browser, que levará para qualquer usuário um gerenciador de aplicações na nuvem, podendo continuar sua utilização de qualquer plataforma sem ultrapassar os limites financeiros do usuário.

Tem-se como pergunta de pesquisa como realizar a implantação do sistema que permita salvar o status de serviço do usuário possibilitando a sua continuação em outro momento de outra plataforma utilizando Docker, uma plataforma livre para desenvolvedores e administradores de sistemas tornando possível construir qualquer aplicação, em qualquer linguagem usando qualquer ferramenta, sendo assim portável além de poder rodar em qualquer lugar.

Entre nossos objetivos temos em vista disponibilizar aplicações para o usuário na nuvem possibilitando assim ao usuário a não instalação de softwares em suas máquinas diminuindo assim o uso de espaço físico e possibilitando o acesso ao sistema em qualquer lugar. Desenvolver uma aplicação computação em nuvem que possibilite a integração de usuários com qualquer aplicação na nuvem.

Visamos também o desenvolvimento de uma aplicação em que sua arquitetura é a de sistemas que utilizem proxy a gerenciar todas as requisições de usuários no sistema em nuvem para uso cotidiano a fim de encontrar a melhor maneira de atender as reais necessidades dos usuários.

Desenvolvimento de uma aplicação em nuvem que virtualize apenas uma aplicação ao invés de máquinas inteiras, tornando assim sua taxa de uso pequena, pelo fato de requerer menos espaço no HD e de inicialização rápida além de permitir a múltiplos usuários utilize um mesmo sistema em diferentes máquinas físicas sem precisar da sua real instalação.

A pesquisa se dará através da combinação de várias formas de pesquisa; Pesquisa bibliográfica através de livros, artigos, sites e trabalhos de graduação.

Estudo de caso do CoreOS, Node.js e Docker será feita através de livros e principalmente de artigos publicados. Por cloud computing se tratar de um tema relativamente novo, não encontra-se muitos livros disponíveis.

Documentação de projeto e implantação através da metodologia.

O documento está dividido em 6 capítulos. No primeiro capítulo apresenta-se o projeto, expondo uma breve contextualização e apresentando a problemática vislumbrada, assim como os objetivos geral e específicos.

No segundo capítulo é realizada uma revisão sobre o Sistema Operacional utilizado, a Distribuição do Sistema Operacional utilizado e a Plataforma de Desenvolvimento.

O terceiro capítulo faz uma revisão da linguagem de desenvolvimento que foi implementada no projeto e exemplos de implementação da linguagem.

O quarto capítulo faz uma revisão de como criar ou utilizar servidores Proxy.

O quinto capítulo propõe uma introdução sobre o que é Cloud e apresenta a arquitetura do sistema e o projeto implantado.

O sexto capítulo apresenta e discute os resultados obtidos assim como as possibilidades de análise considerando a proposição do trabalho.

2 Unix

O Unix é um sistema operacional, um conjunto de programas projetados para controlar as interações das funções de baixo nível com as aplicações (programas) e com os usuários. O Unix controla os recursos do computador, distribui as tarefas entre os diferentes usuários, controla a ordem das tarefas.

As raízes do Unix tiveram início nos anos 60 quando Thompson, Dennis Ritchie e outros desenvolvedores se juntaram para desenvolver o sistema operacional Multics nos Laboratórios Bell da AT&T. A ideia era criar um sistema capaz de comportar centenas de usuários, mas diferenças entre os grandes grupos envolvidos na pesquisa (AT&T, General Eletronic e Instituto de Tecnologia de

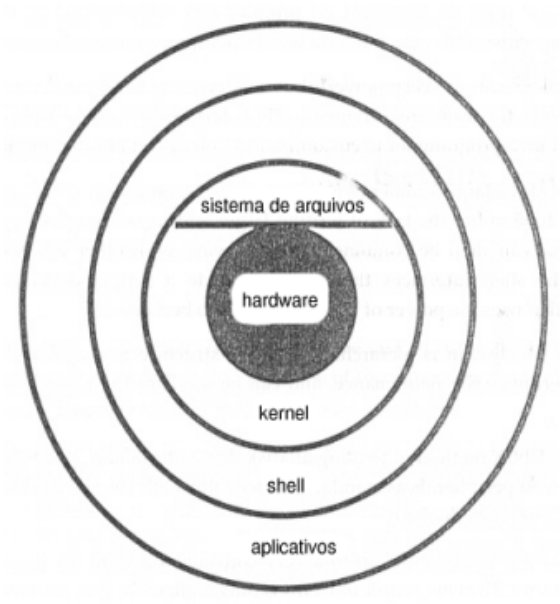
Massachusetts) levaram o Multics ao fracasso. Contudo, em 1969 de acordo com o site oficial, Thompson começou a reescrever o sistema com pretensões não tão grandes, e aí surge o Unics (INTRODUÇÃO, s.d).

A linguagem empregada no sistema passa a ser a C (RITCHIE & THOMPSON, 1974), algo apontado como um dos principais fatores de sucesso do sistema. Atualmente, uma série de Sistemas Operacionais são baseados no Unix, entre eles, nomes consagrados como Gnu/Linux, Mac OS X, Solaris e BSD.

Sendo um sistema multitarefa, capaz de executar dezenas de processos simultaneamente. Outra característica do Unix é o suporte a multiusuário. O sistema permite que várias aplicações sejam executadas de modos independente e concorrente por usuários diferentes. Assim, eles podem compartilhar não somente hardwares, mas também softwares e componentes como discos rígidos e impressoras.

O sistema operacional Unix é composto por quatro partes básicas, como ilustrado na figura 1.

Figura 1: Quatro partes básicas do Unix



Fonte: (INTRODUÇÃO, s.p).

Unix é um sistema proprietário, justamente por isso softwares como distribuições de Linux e o Mac OS são chamadas de “tipo Unix”. Para um sistema ser considerado Unix, é preciso se enquadrar completamente no “Single Unix Specifications” ou Especificações Únicas do Unix, em tradução livre.

Se tomarmos Windows, Mac e Linux, os principais sistemas operacionais da atualidade, apenas o sistema operacional da Microsoft não faz uso de uma arquitetura baseada no Unix.

2.1 Kernel

O termo Kernel vem do inglês, e significa “núcleo”. Apesar de pouco comentado no ambiente da informática, o Kernel possui papel muito importante para o funcionamento de um computador. O Kernel é considerado o principal item dos sistemas operacionais, sendo que ele é a ligação entre o processamento de dados e os programas. Por isso, muitos o consideram o cérebro do computador. O Kernel ganhou notoriedade com o desenvolvimento do Linux, porém ele também está presente no Windows e no Mac OS.

O Kernel é o grande responsável por fazer a ligação entre o hardware e o software do computador (FISHER, 2008). Sendo assim, o objetivo principal é gerenciar a máquina e fazer com que os aplicativos possam ser executados através dos recursos existentes no computador. Além disso, o Kernel tem como responsabilidade garantir que a memória RAM seja utilizada do melhor modo possível para que assim não ofereça qualquer risco para o computador.

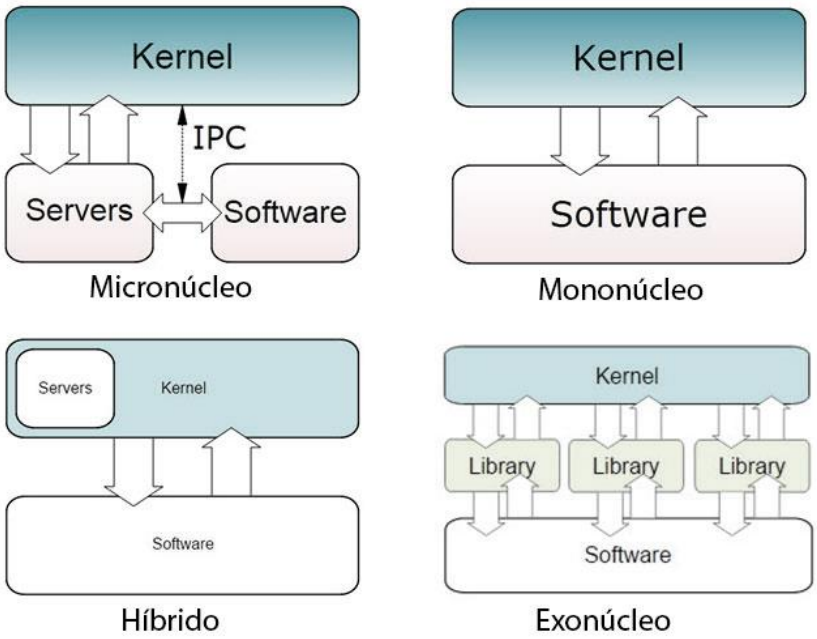
Ao ligar um computador, o Kernel é imediatamente acionado e começa a detectar o hardware que a máquina possui e ainda o que precisa para prosseguir em sua função. O núcleo então, após o sistema operacional ser carregado, possui função também de gerenciar outras questões, como os

arquivos, memórias, entre outros, tudo isso para garantir a organização funcional do sistema.

Além disso, o Kernel pode decidir quais dos programas que estão sendo executados no momento que devem ser alocados para o processador, ou mesmo processadores.

Em resumo, o Kernel é o grande responsável por gerenciar os recursos do sistema e assim, permitir que os programas possam fazer uso deles. O funcionamento não se dá de forma simples, é sim um processo complexo, dependendo do tipo de Kernel que sua máquina possui. Ele pode ser dividido como no exemplo da figura 2.

Figura 2: Os quatros tipos de Kerne.l



Fonte: (POZZEBON, 2015).

Monolítico: Neste caso, os controladores de dispositivos e também as extensões de núcleo são executadas no espaço de núcleo, tendo acesso total ao hardware.

Micronúcleo: Alguns dos processos são executados no próprio núcleo, porém, o restante pode ser executado no espaço vago. Permite alternar

dinamicamente entre sistemas operativos e manter mais de um deles ativos simultaneamente.

Híbrido: É considerado um micronúcleo e conta com um código no espaço do núcleo para que as operações executadas possam ser mais velozes.

Nanonúcleo: Delega virtualmente todos os serviços para os drivers de dispositivo, desde os mais simples, como um temporizador. Com isso torna o requerimento de memória do núcleo ainda menor do que o dos micronúcleos.

Exonúcleo: Este tipo de núcleo aloca recursos físicos de hardware, podendo, por exemplo, fazer que um programa sendo executado em um exonúcleo possa se ligar com uma biblioteca do sistema que também usa exonúcleo para fazer simulações do sistema.

2.2 Linux

Linux é ao mesmo tempo um Kernel e o sistema operacional que roda sobre ele. De acordo com CAMPOS (2006) o Kernel Linux foi criado em 1991 por Linus Torvalds, então um estudante finlandês, e hoje é mantido por uma comunidade mundial de desenvolvedores, que inclui programadores individuais e empresas como a IBM, a HP e a Hitachi, coordenada pelo mesmo Linus, integrante da Linux Foundation.

O Linux adota a GPL, uma licença de software livre - o que significa, entre outras coisas, que todos os interessados podem usá-lo e redistribuí-lo, nos termos da licença. Aliado a diversos outros softwares livres, como o KDE, o GNOME, o Apache, o Firefox, os softwares do sistema GNU e o OpenOffice.org, o Linux pode formar um ambiente moderno, seguro e estável para desktops, servidores e sistemas embarcados (CAMPOS, 2006).

O sistema funciona em dezenas de outras plataformas, desde mainframes até relógios de pulso, passando por várias arquiteturas: Intel, StrongARM,

PowerPC, Alpha etc., com grande penetração também em dispositivos embarcados, como handhelds, PVR, videogames e centrais de entretenimento - nos quais há expoentes como o sistema Android, mantido pelo Google.

Muitos conhecem e divulgam o sistema operacional do pinguim apenas como Linux, porém o termo correto é GNU/Linux. Em palavras simplificadas, Linux é apenas o Kernel do sistema operacional, ele depende de uma série de ferramentas para funcionar, a começar pelo programa usado para compilar seu código-fonte. Essas ferramentas são providas pelo projeto GNU, criado por Richard Stallman (CAMPOS, 2006).

A vantagem de um sistema de código aberto é que ele se torna flexível às necessidades do usuário, tornando assim suas adaptações e correções muito mais rápidas. O código-fonte aberto do sistema permite que qualquer pessoa veja como ele funciona, corrija algum problema ou faça alguma sugestão sobre sua melhoria.

O Linux possui várias distribuições. Uma distribuição nada mais é que um Kernel acrescido de programas escolhidos a dedo pela equipe que a desenvolve. Cada distribuição possui suas particularidades, tais como forma de se instalar um pacote, interface de instalação do sistema operacional em si, interface gráfica, suporte a hardware. Restando ao usuário definir que distribuição atende melhor suas necessidades..

A linha de comando é o método mais usado por administradores de sistemas Linux, pois é o que oferece o maior número de possibilidades, além de ser o método mais rápido de fazer as coisas.

2.3 CoreOS

CoreOS é uma distribuição livre Linux nova e leve baseada em Linux Kernel que foi projetada para prover infraestrutura para implementações clusters

capaz de rodar em qualquer plataforma que utiliza containers Linux para lidar com serviços num nível de abstração alta. Nele, um serviço e todas suas dependências - que, em Linux são associadas com pacotes de gerenciamento - são agrupadas dentro de um container que pode ser rodado em uma ou várias máquinas CoreOS.

Com isso, o CoreOS intencionalmente não inclui um pacote de gerenciamento, pois sua filosofia é de que qualquer software dentro dele deve ser executado dentro de um container, usando bem menos memória do que qualquer distribuição Linux típica.

O CoreOS de fato é um sistema operacional bem leve, pois ele oferece apenas o mínimo de funcionalidade requerida para implementar aplicações dentro de seus containers junto com mecanismos incorporados (YEGULALP, 2014).

É preciso salientar que seus containers Linux providenciam benefícios similares à de uma máquina virtual completa, mas que apenas focam em aplicações ao invés de hosts virtualizados. De acordo com BOND (2014), a maioria das distribuições em Linux vem com um certo número de funções que a maioria das implementações de softwares não usam, tornando-se assim mais complexas e pesadas.

Além disso, uma parte do CoreOS importante de salientar é que ele também utiliza containers para lidar em como serviços e aplicativos implantados em um cluster rodam, podendo-se assim fazer que todas as máquinas que rodam no cluster compartilharem um único sistema de inicialização.

No nível de desenvolvimento de aplicações web a opção de usar tal distribuição torna-se ainda mais justificável ao pensar que o usuário final não precisará ter todas suas dependências em um nível do sistema operacional e sim apenas colocá-las dentro de um container e instalar seus containers para a aplicação web.

Em nosso projeto, estaremos utilizando tal distribuição, pois ela facilmente lida com dependências, ser mais leve para rodar aplicações dentro de containers na nuvem, atendendo assim um dos maiores requisitos para o projeto.

CoreOS roda em quase qualquer plataforma, incluindo Vagrant, Amazon EC2, QEMU / KVM, VMware e OpenStack e seu próprio hardware. É projetado para segurança, consistência e confiabilidade. Em vez de instalar pacotes via yum ou apt, CoreOS utiliza um Linux container para gerenciar seus serviços em um nível mais alto de abstração. O código de um único serviço e todas as dependências são armazenados dentro de um container que pode ser executado em uma ou várias máquinas CoreOS.

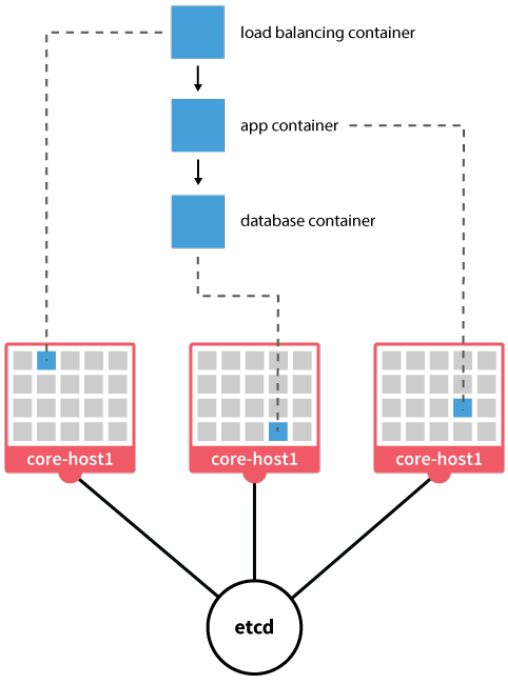
Containers Linux oferecem benefícios semelhantes como máquinas virtuais completas, mas focada em aplicações em vez de todos os hosts virtualizados, não tendo quase nenhuma sobrecarga de desempenho. A falta de sobrecarga permite que você ganhe densidade, o que significa menos máquinas para operar e um gasto menor de computação.

O principal bloco de construção de CoreOS é Docker, um Linux container, onde suas aplicações executam o código. Docker é instalado em cada máquina CoreOS. É preciso construir um container para cada um de seus serviços (servidor web, cache de banco de dados) iniciá-los com fleet e conectá-los através da leitura e escrita para ETCD.

A maneira recomendada para executar o Docker em seus CoreOS é com fleet, uma ferramenta que apresenta todo o seu conjunto como um único sistema de inicialização. Fleet trabalha por recebimento de arquivos da unidade systemd e gerencia em máquinas do cluster com base em conflitos declarados e outras preferências armazenadas na unidade de arquivos.

Fleet pode implantar serviços de alta disponibilidade, garantindo que containers de serviço não estão localizados na mesma máquina, exemplo figura 3, zona de disponibilidade ou região. Fleet também suporta a partilha as mesmas propriedades.

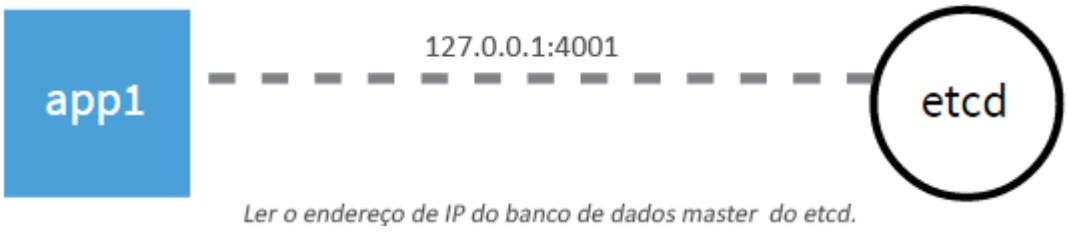
Figura 3: Executando serviços com Docker



Fonte: (COREOS, 2014, s.p).

Cada host fornece um ponto de extremidade local para ETCD, que é usado para a descoberta de serviços e valores de configuração de leitura/escrita. Este ETCD é replicado e todas as alterações são refletidas em todo o cluster. Sua aplicação pode, por exemplo como na figura 4, sempre contatar a instância ETCD local em 127.0.0.1:4001.

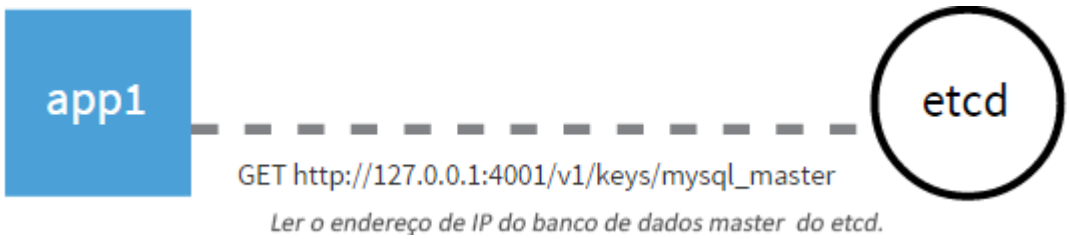
Figura 4: Ler o endereço de IP do banco de dados master do etcd.



Fonte: (COREOS, 2014, s.p).

Digamos que estamos executando uma aplicação web simples, como Wordpress. Em vez de codificar o nosso banco de dados com endereço no arquivo de configuração, vamos buscá-lo a partir de ETCD. É tão simples como http://127.0.0.1:4001/v1/keys/database e utilizando a resposta dentro de seu código de conexão database, segundo o exemplo da figura 5.

Figura 5: Acessando banco dados por IP.



Fonte: (COREOS, 2014, s.p).

Um exemplo mais avançado envolve a descoberta de serviços para um proxy. Cada um de seus containers com aplicativo pode anunciar-se a ETCD após o início, e seu proxy está aguardando as mudanças no mesmo valor. Agora, o container de proxy automaticamente sabe quando se reconfigurar e novos containers recebem automaticamente o tráfego. Usando a descoberta de serviços em seu aplicativo permite que você adicione mais máquinas e escale seus serviços de forma integrada.

CoreOS fornece estável e confiável atualizações para todas as máquinas. Todas as atualizações são realizadas offline a partir do processo de construção, e os dados de cada atualização são transmitidos para a máquina através de um certificado SSL. Estes dados são verificados em relação a atualização baixada antes de ser aplicada à máquina.

Para proteger todo o cluster é recomendado usar um firewall físico, grupos de segurança ou um recurso semelhante para restringir todo o tráfego. Se você estiver executando containers que são utilizados para balanceamento de carga ou cache, considere expor as portas para esses containers em vez de todos os recipientes. Você ainda pode automatizar esse processo com ETCD.

2.4 Docker

De acordo com Hykes (2014), Docker é uma plataforma aberta para desenvolvedores e administradores de sistemas para a criação e execução de

aplicações distribuidas. Ela é composta pela Docker Engine e Docker hub. Docker Engine é uma ferramenta de tempo de execução e pacote portatil leve, já o Docker Hub trata-se de um serviço de nuvem usado para compartilhar aplicativos e automatizar fluxos de trabalho.

Docker permite que aplicações sejam rapidamente montadas a partir de componentes e elimina o atrito entre os ambientes de desenvolvimento, controle de qualidade e produção. Como resultado é possivel enviar mais rapido e executar o mesmo aplicativo inalterado em laptops, data centers virtualizados e em qualquer nuvem.

Segundo Martins (2014), através do Docker desenvolvedores podem construir qualquer aplicativo em qualquer idioma usando qualquer conjunto de ferramentas. Sendo totalmente portateis, os apps "Dockerized" podem ser executados em qualquer lugar. Exemplos: OS X, Windows, servidores de controle de qualidade que executam o Ubuntu na nuvem, e máquinas virtuais rodando Red Hat.

Os desenvolvedores podem começar a desenvolver usando qualquer um dos mais de 13.000 aplicativos disponíveis em Docker Hub. Docker gerencia e controla as alterações e dependências, tornando mais fácil para os administradores de sistemas a entender como os aplicativos de trabalho são construidos. E com Docker Hub, os desenvolvedores podem automatizar suas linhas de comando e compartilhar partes de projetos com os colaboradores através de repositórios públicos ou privados.

Docker ajuda administradores a implantar e executar qualquer aplicativo em qualquer infraestrutura, de forma rápida e confiável. Fornecendo para administradores de sistemas ambientes padronizados para suas equipes de desenvolvimento, controle de qualidade e produção, reduzindo "trabalho na maquina real". Por "Dockerizing" a plataforma de aplicativo e suas dependências, os administradores de sistemas podem abstrair diferenças do sistema operacional distribuido e infraestrutura subjacente.

Além disso, a padronização na Docker Engine como unidade de implantação dá aos administradores flexibilidade nas cargas de trabalho executado. Quer

em data center virtualizados ou nuvens públicas, implantação de carga de trabalho é menos limitado pela tecnologia de infra-estrutura e em vez disso é orientado para as prioridades e políticas de negócios. Além disso, a execução leve do Docker Engine permite escalonamento rápido em resposta às mudanças na demanda.

Docker compreende apenas a aplicação e suas dependencias, sendo executado como um processo isolado no espaço do sistema operacional do usário hospedeiro, compartilhando o Kernel com outros recipientes. Tendo os mesmos beneficios de recursos de uma maquina virtual, porém muito mais eficiente e portátil.

Segundo Lowe (2014), um container Docker - atualmente utilizando Linux Containers (LXC) - se destina a executar uma única aplicação, criando containers altamente específicos destinados a executar o MySQL, Nginx, Redis, ou algum outro aplicativo.

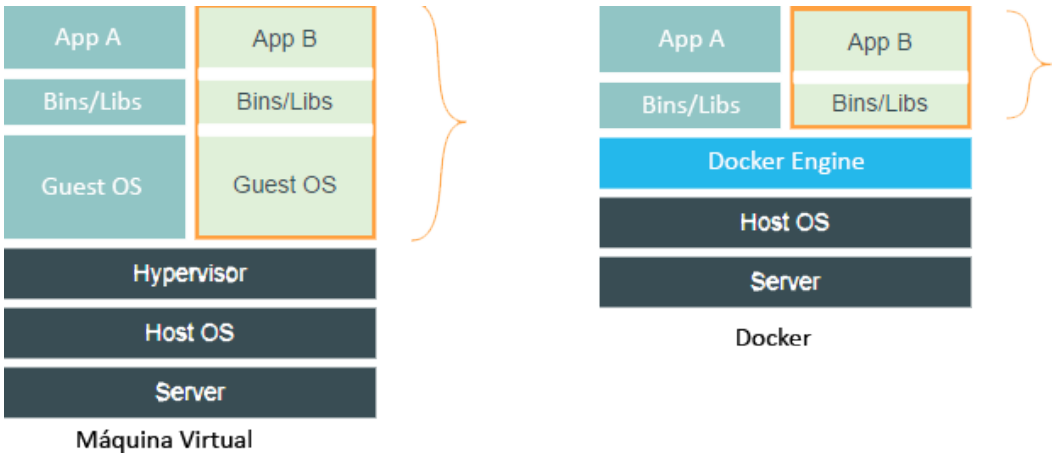
Quando é preciso executar dois aplicativos ou serviços distintos em um ambiente Docker é possível executá-los juntos em um único container ou pode-se usar dois containers separados, o recomendável. O Docker tem um escopo específico. É projetado para a aplicação não tendo como escopo a uma aplicação específica. Em vez disso, LXC tem como escopo a uma instância do Linux. Da mesma forma, as tecnologias container baseados no Windows são escopo para uma instância do Windows.

A máquina virtual, por outro lado, tem um alcance mais amplo. Ele tem como escopo a qualquer sistema operacional suportado. Ao usar a virtualização completa-máquina, você não está limitado a apenas Linux ou Windows somente, podendo executar praticamente qualquer versão recente do sistema operacional.

Um sistema totalmente virtualizado normalmente leva alguns minutos para começar, containers LXC leva segundos, e às vezes até menos de um segundo.

Outra diferença da máquina virtual está na utilização do espaço, segue exemplo da figura 6. Máquinas virtuais consomem uma grande quantidade de CPU e memória, podendo aumentar o tamanho da máquina virtual ao tentar lidar com todas as dependências e pacotes necessários. Digamos que você tem uma imagem de recipiente que é de 1GB de tamanho. Se você quiser usar uma máquina virtual completa, você precisa ter 1GB vezes x número de Máquinas virtuais que você quer. Com Docker você pode compartilhar a maior parte do 1GB e se você tem 1000 containers você ainda pode ter apenas um pouco mais de 1 GB de espaço para a utilização dos containers, assumindo que todos tenham a imagem do mesmo sistema operacional em execução.

Figura 6: Comparação entre Máquinas Virtuais e Docker



Fonte: (COREOS, 2014, s.p).

Como podemos ver na foto, além de incluir a aplicação e suas bibliotecas, na máquina virtual é requerido o uso de um sistema operacional cliente, transformando o que se poderia ser originalmente em poucos megabytes em gigabytes.

Outra desvantagem de utilizar a máquina virtual, do ponto de vista de um desenvolvedor, as ferramentas para a criação de aplicativos e testes são limitados. Além de produzir uma sobrecarga significativa de desempenho, especialmente ao executar operações de entrada e saída.

LXCs são leves e permitem executar várias instâncias isoladas no mesmo host. Eles compartilham um único núcleo, mas pode ter uma definição de conjunto

para o número de recursos que pode consumir. LXCs permiti a circulação segura de casos isolados sem interferência entre essas instâncias.

Docker é uma camada adicional de LXCs, para torná-los mais fáceis de usar em operações mais complexas. Tendo como diferencial:

1. Versão com rastreabilidade completa dos estados de produção em servidor de volta para o desenvolvedor container real;
2. Ele ajuda a evitar a dependência, fornecendo um isolamento completo de recursos, rede e conteúdo;
3. Constrói containers facilmente compartilháveis de forma incremental;
4. Suporta um ecossistema de compartilhamento de imagem;
5. Sua reutilização container permite criar componentes mais especializados;
6. Usa primitivas do sistema operacional;
7. Ele fornece uma linha distinta de separação de funções que torna a vida mais fácil para os desenvolvedores

3 Node.js

Node.js é um contexto JavaScript orientado a eventos que permite rodar códigos em JavaScript no backend, ou seja, um ambiente de tempo de execução simples para JavaScript. Para clarificar, o JavaScript necessitaria de um interpretador para ser então executado no backend e isso é o que o Node.js faz.

De acordo com Cannaday (2013), é importante ressaltar que Node.js não é um web server, pois ele por si só não faz nada, por exemplo, caso seja necessário

controlar o servidor HTTP, o desenvolvedor deveria escrever seu próprio servidor HTTP com ajuda de bibliotecas inclusas.

Node.js é leve e eficiente, tornando-se ideal para aplicações em tempo real com um grande número de dados trafegando. Seus desenvolvedores afirmam que ele jamais dará deadlock, pois ao invés de utilizar abordagem multi-thread comuns ele usa um modelo de chamadas E/S assíncronas com uma programação orientada a um evento, além de não bloquear E/S se tornando assim recomendado para sistemas escalonáveis.

Tal modelo de chamadas E/S faz com que nenhuma tarefa de leitura e escrita trave sua aplicação pois a mesma será rodada em background, ou seja, de forma assíncrona, e quando as mesmas forem finalizadas seus resultados serão exibidos por um evento call-back das funções, enquanto o resto das funções continuam rodando

O Node.js roda dentro de uma JavaScript V8 VM, que é um motor usado pela Google no Chrome, que compila JavaScript em código de máquina antes da execução, que de acordo com (GLOVER, 2011):

> *[...] resulta em desempenho extremamente rápido no tempo de execução — algo que não é geralmente associado a JavaScript.*

Dessa forma Node permite desenvolver sistemas extremamente rápidos e altamente simultâneos.

Um ponto importante do Node.js para ressaltar é que, por ser assíncrono, não há como determinar a ordem de execução das chamadas efetuadas, porém uma série de módulos foram desenvolvidos pela comunidade afim de manter o controle da ordem de execução sem que se peca a legibilidade para que sua futura manutenção do código seja de fácil compreensão.

3.1 JavaScript

JavaScript é uma linguagem de programação para HTML e Web comumente utilizado para lidar com comportamentos de páginas web. A integração do JavaScript no meio dos ano 90 tornou muito mais fácil para desenvolvedores acessarem elementos HTML, tornado-o muito famoso para controlar, customizar e adicionar animações.

No JavaScript, o tipo de dados core do é justamente o tipo Object, sendo este o mais comumente usado e mais fundamental, sendo que este possui cinco tipos primitivos: Number, String, Boolean, Undefined e Null, sendo que os primitivos são imutáveis e objetos são mutáveis.

Um objeto é uma lista não organizada de tipos primitivos,que são armazenados por pares de "nome-valor". Cada item na lista é chamada de propriedade. Um adento é que funcções são chamadas de métodos.

Na Listagem 1, temos um exemplo de um objeto simples, onde armazenaremos o máximo de convidados de um casamento por parte da noiva e do noivo.

```
var convidados = {noiva: "130", noivo: "130"};
```

Listagem 1 – Lista de objetos

O objeto "convidados" é uma lista que contém itens e cada item possui uma lista que é salva no formado de "nome-valor". Os items que podemos verificar no objeto criado acima são "noiva" e "noivo", e seus valores "130" cada.

Porém um Object não possui apenas propriedades mas também atributos, ou seja, cada objeto não possui apenas pares de "nome-valor", mas também três atributos que são armazenados como verdadeiros por padrão, sendo eles:

- Configurable Attribute: Especifica se a propriedade pode ser deletada ou alterada.
- Enumerable: Especifica se a propriedade pode ser retornada em um loop.
- Writable: Especifica se a propriedade pode ser alterada.

Por mais que muitos autores digam que no JavaScript tudo é um objeto, isso não é uma realidade para todos os casos. Por exemplo, alguns pensam que funções e objetos são coisas distintas, porém no JavaScript uma função é um objeto, funções de objetos de classes.

Quando pensamos em números, strings e booleans na maioria das linguagens de programação, pensamos eles como tipos primitivos, ou seja, eles não são um objeto. Porém no JavaScript eles são considerados objetos.

No JavaScript o Garbage Collector pode geralmente atrapalhar a performance de uma aplicação a cada chamada do evento.

Garbage Collector significa uma forma de gerenciar memória dinamicamente/automaticamente. Onde ele tenta "recolher lixo do programa", ou seja, recupera o espaço alocado em memória para determinado objeto que não é mais utilizado pelo programa. Liberando assim mais espaço para o programa ser executado. É comum que em alguns casos, os programadores tenham que especificar quais objetos serão ou não desalocados e retornados para a memória do sistema. No exemplo para os que não devem ser desalocados, objetos que fazem parte de uma classe e que no futuro seja necessária a utilização dele.

3.2 ChromeV8

ChromeV8 é um motor para JavaScript open source escrito em C++ e usado no Google Chrome, o navegador open source do Google. O V8 é uma implementação do ECMAScript especificado no ECMA-262, que funciona tanto para sistemas operacionais Windows, Mac OS X e Linux. O V8 utiliza processadores IA-32, X64 ou ARM para rodar.

O V8 pode ser usado tanto em um navegador (principalmente Chrome e Chromium) ou como um motor independente de alta performance que pode ser

integrado à projetos independentes, como no caso o server-side Node.js, ou no cliente-side como o .NET/Mono usando V8.NET.

Simplificando o V8 compila e executa código em JavaScript no server-side. Diferente do que estamos acostumados com o JavaScript tradicional, onde seu código funciona no client-side, ou seja, no navegador do cliente.

De acordo com (Google, 2015), desenvolvedor do motor, o JavaScript é gerencia a alocação de memórias para objetos e libera os objetos coletados que não são mais necessários, sendo a perfomance principal dele de "para o mundo, generalizar e preciso" coletor de lixo.

A perfomance padrão do JavaScript é muito limitada, o que tornou um fator decisivo para o desenvolvimento de aplicações Web. Para contornar a situação, o Google desenvolveu o motor que executa rapidamente códigos JavaScript, mais rápido do que o JScript usado no Internet Explorer, SpiderMonkey utilizado no Mozilla Firefox e do que o JavaScriptCore, utilizado no Safari. Tornando o V8, talvez a melhor opção para aplicações que necessitem muito de JavaScript para sua perfeita performance de execução.

As três áreas chaves para a perfomance do V8, segundo (Google, 2015) são:

- Acesso rápido às propriedades
- Geração de Código de Máquina Dinâmico
- Garbage Collector Eficiente

3.2.1 Acesso rápido às propriedades

Na maioria dos motores JavaScript, o acesso a objetos na memória torna-se mais lenta do que em comparações com linguagens de programas conhecidos, onde o acesso é determinado pelo compilador que fixa um espaço para o objeto baseado na classe do objeto. Em outros motores JavaScript porém, o

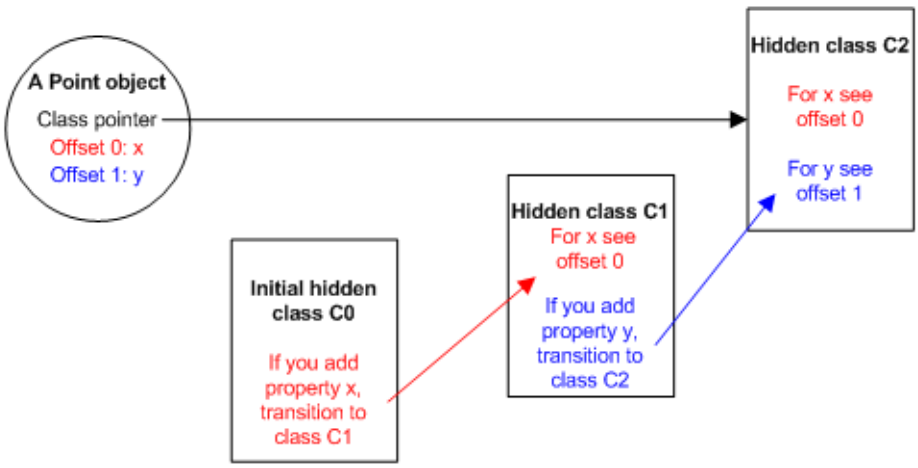
acesso é por meio de uma biblioteca que armazena dinamicamente a propriedade do objeto na memória. Com isso, o acesso à memória torna-se mais lenta do que simplesmente ler ou armazenar valores, tendo sempre que procurar onde a propriedade foi armazenada.

Visando isso, o motor da Google não usa essa biblioteca dinâmica, mas sim a criação de "classes secretas", onde um objeto muda sua classe secreta quando uma nova propriedade é adicionada. Basicamente, na primeira criação de um objeto, cria-se uma classe secreta, caso ele não possua propriedades, o padrão é essa classe secreta estar vazia.

Ao criar uma nova propriedade para o objeto, cria-se uma nova classe secreta que baseia-se na primeira classe secreta criada e adiciona informações à ela sobre a nova propriedade criada. Para cada nova propriedade de um objeto, a nova propriedade baseia-se na propriedade criada anteriormente, criando assim uma árvore de propriedades de cada objeto. Durante a criação de cada propriedade, o V8 desloca um espaço para a propriedade criada e cada propriedade é uma transição para outra que se será criada ou não.

No exemplo abaixo podemos ter uma noção melhor sobre essa transição e como cada classe secreta se baseia, onde o objeto Point possui duas propriedades, chamadas x e y, essa por sua vez cria uma classe secreta chamada de C0 e cada propriedade possui uma referência de transição para a próxima propriedade, conforme Figura 7:

Figura 7: Exemplo de classes secreta do V8.



Fonte: (GOOGLE, s.d)

Dessa forma, conforme Figura 7, a localização de cada propriedade torna-se sequencial, o que em tempo de execução diminui significamente o tempo de resposta de acesso, tornando o V8 uma opção[1] para sistemas que são fortemente baseados em JavaScript.

[1] Benchmarking do V8, disponível em: http://benchmarksgame.alioth.debian.org/u64/javascript.html

3.2.2 Geração de Código de Máquina Dinâmico

No V8, o código JavaScript é compilado na máquina de código quando executado pela primeira vez, sem a necessidade de um interpretador. No início de uma execução, o V8 aloca classes secretas para o objeto atual, otimizando o acesso à leitura de suas propriedades, predizendo que todos os futuros objetos serão acessados na mesma seção de código no cache e que usa informação da classe usarão essas classes secretas, ou seja, todos os objetos se basearão numa classe base e essa possuirá informação para a próxima transição de suas propriedades.

Dessa forma, ao acessar uma certa propriedade, o V8 verifica na cache a classe secreta do objeto e ao encontrar, retornar o valor da propriedade, também caso o valor na cache seja diferente da classe secreta, ele incorpora o

código na cache. Dessa forma é optimizada a velocidade de criação de objetos com mesma estrutura de classe secreta e leitura de suas respectivas propriedades, onde procura-se na cache incorporada da classe secreta e executa o código em seu motor.

3.2.3 Garbage Collector Eficiente

O garbage collector do V8 é um coletor incremental geracional. O assembler dele é baseado no assembler do Strongtalk. Seu coletor de lixo, recupera a memória alocada por objetos que não estão mais em uso, ou que não são mais necessárias. Um coletor é um ponto muito importante em qualquer sistema, onde à cada verificação do coletor, mais espaço em memória é liberada para que outros futuros objetos possam utilizar aquele espaço. Assegurando assim uma alocação rápida de objetos em memória.

O motor V8, para assegurar otimização de tempo de execução do JavaScript, pausa o sistema quando está realizando um ciclo e processa apenas uma parte do heap do objetos, minimizando o impacto da pausa do sistema. Por utilizar classes secretas, o motor sempre sabe onde estão os objetos e seus ponteiros na memórias.

Durante seu ciclo, o heap de um objeto é dividido em duas partes: uma onde aloca-se espaço para objetos que são criados além de um espaço antigo onde objetos que sobreviveram à um ciclo são adicionados. Caso durante um ciclo do coletor de lixo um objeto é movido, o V8 automaticamente atualiza todos os ponteiros para o objeto.

3.3 Express.js

O Express.js é um framework para Node.js, modelo para aplicações web de única-página, multi-página e hibridas que pega um Node da aplicação e o transforma em algo que se comporta como web servers com mais funcionalidades. O Express.js é utilizado em sites como Walmart, Paypal e Netflix.

O framework ajuda ao programador a organizar sua aplicação web e uma arquitetura MVC server-side. Com ele, é possível gerenciar rotas, requests e views.

Com o Express.js é possível criar um projeto com a arquitetura MVC tradicional ou pode-se usar rotas que são criadas, que são como uma combinação de modelos e controles que podem possuir configurações e até lógica dentro deles.

Enquanto o Node.js puro pode ser complexo, o Express.js organiza Node no seu server-side para módulos gerenciáveis e testáveis. Facilitando assim a construção da aplicação, pois o desenvolvedor se concentra no que a aplicação faz ao invés de detalhes téncicos.

No Express, quando se cria um webserver, escreve-se uma função JavaScript que funciona para a aplicação inteira, essa que por si fica lendo requisições de navegadores, aplicações mobiles que consomem a API criada, ou qualquer ou cliente que fale com o servidor criado. Quando um request é identificado, a função identifica ela e já decide como respondê-la. Exemplo, caso um request venha da página principal, no nosso exemplo ele será a página http://localhost/, responderemos como uma mensagem de boas vindas.

```
app.get('/', function (req, res) {
  res.send('Seja bem vindo!');
});
```

Listagem 2 – Resposta de um Request

Na listagem 2 demonstramos o que é chamado de Routing no Express, que significa como a aplicação se comporta ao receber uma requisição de uma URI com qualquer método HTTP (POST, GET, HEAD, PUT, DELETE, TRACE

entre outros) de um cliente para um determinado endpoint. Cada rota suporta um ou mais de uma função que serão executadas.

O Routing lê a estrutura do app.METHOD(PATH, HANDLER),que por si é uma instância do express e METHOD são os métodos requests do HTTP. Path é o caminho no servidor e HANDLER é a função a ser executada quando a condição é verdadeira.

No exemplo, podemos ver que acessar a página http://localhost/, como não especificamos qual o tipo de método, o navegador reconhece automaticamente a requisição como do tipo GET. Ao verificar na função que o método confere com do tipo GET e que a URI solicitante é a default, ou seja http://localhost/, a função responde enviando uma mensagem de boas vindas.

Caso seja necessário que uma rota aceite todas as requisições de métodos HTTP existentes, para não escrever a mesma rota para métodos diferentes, existe a opção de aceitar todos os métodos, sendo ela o app.all. Na listagem 3 a seguir, qualquer requisição para a rota /novidades será enviada uma resposta de “OK”. Sendo ela:

```
app.all('/novidades', function (req, res, next) {
   res.send('OK');
});
```

Listagem 3 – Resposta para requisições

Ao enviarmos por exemplo uma requisição GET do método HTTP DELETE para a rota /novidades, o Express.js nos retorna uma mensagem de OK. Caso a requisição seja do tipo REQUEST com método HTTP PUT para a mesma rota, o Express.js também nos retorna uma mensagem de OK.

Em suma qualquer request que o cliente solicite ao servidor será redirecionada para qualquer handler escrita pelo desenvolvedor. Onde o servidor enviará o request o a aplicação em Express que por fim tratará seu request em uma função, logo após tratar o request em uma função, sua resposta é enviada diferente ao servidor (Node HTTP) que enviará uma resposta ao cliente.

Um middleware é uma função invocada pela camada de rota do Express antes do último handler do request. No Express uma aplicação é uma série de chamadas para middleware, que por si é uma função com acesso a objetos request e response.

Segundo o guia oficial do Express, quando a função realiza um acesso aos objetos request e response, ela também gerencia o próximo middleware na pilha para o ciclo de resposta, chamado por eles de next caso o atual middleware não finalize o ciclo.

Existem vários de middlewares que uma aplicação pode usar, sendo ela Application-level middleware, Router-level middleware, Error-Handling middleware, Built-in middleware, Third-party middleware.

Um apllication-level middle são instâncias do express usando app.use e app.VERB(). O Router-level funciona similarmente como o application-level, porém eles são instâncias do express.Router(). O Error-handling middleware funciona como qualquer outro middleware, porém ele recebe quatro argumentos ao invés de três, como podemos ver na listagem 4:

```
app.use(function(err, req, res, next) {
      res.status(404).send('Página não encontrada.');
});
```

Listagem 4 – Error-handling

Na listagem 4, caso seja acionada um erro do status 404, ou seja, Not-Found, será enviada uma resposta de página não encontrada..

3.4 PasspostJS

Passport é um middleware de autenticação para o Node.js onde é fácil de gerenciar os sucessos e erros, além de possuir mais de 140 estratégias de autenticação que podem ser escolhidas, permitir implementação de estratégias customizadas.

O Passport serve para autenticar requests, facilitando o encapsulamento dos módulos, delegando todas as outras funcionalidades para a aplicação. Separando do código em si, facilitando a manutenção da aplicação, além de facilitar a integração na aplicação.

Passport pode ser usado para autenticações locais ou até mesmo em autenticações que integrem com várias APIs externas, como por exemplo o Facebook, Twitter.

As estratégias no Passport são mecanismos de autenticação empacotadas em módulos. A aplicação pode escolher qual estratégica utilizar sem a criação de dependências desnecessárias. No passport, informamos quais estratégias gostaríamos de usar e então implementar o middleware Passport na aplicação, assim o Passport ficará responsável por todas as autenticações necessárias.

Para autenticar um request, basta chamar o passport.authenticate() e informar qual estratégia será implementada. No exemplo abaixo será mostrado como criar uma estratégia de autenticação do tipo local, onde essa estratégia receberá um nome de usuário, senha e verificará se existe no banco de dados um usuário com tal dados.

```
passport.use(new LocalStrategy(function(username, password,done){
    Users.findOne({ username : username},function(err,user){
        if(err) { return done(err); }
        if(!user){
            return done(null, false, { message: 'Usuário inexistente.'
});
        }
        hash( password, user.salt, function (err, hash) {
            if (err) { return done(err); }
            if (hash == user.hash) return done(null, user);
            done(null, false, { message: Senha inválida.' });
        });
    });
}));
```

Listagem 5 – Estratégia de autenticação local

Com a listagem 5, podemos verificar que após receber os dados do request, com usuário e senha, a estratégia buscar por um usuário pela função Users.findOne, onde *Users* pode ser, por exemplo, um modelo de um Schema de banco de dados para Usuários. Caso ocorra algum erro durante a busca, a autenticação retornará o objeto *done* com os dados do erro cusador. Caso não encontre um usuário com tal nome informado, retornará o objeto *done* com uma mensagem customizada. Caso encontre o usuário e não encontre nenhum erro, criará um hash da senha, caso ocorra algum erro, retornará os dados do erro e caso seja igual com a informada pelo request, retornará o usuário com suas informações, caso a senha seja incorreta, uma mensagem de erro será disparada.

Para consumir a autenticação, como na listagem 5, usaremos um formulário apontando para uma URI de login através do método POST de HTTP. Essa URI será interceptada através das rotas informadas no Express.js e então nelas será feita uma requisição para o Passport validar as informações. A tag simples de form conterá no fim as seguintes informações:

```
<form action="/login" method="post"> Dados do login... </form>.
```

Listagem 6 – Tag Form

A rota no Express.js ficará escutndo qualquer requisição POST com o caminho “/login”, após validar que o informado no request coincide com a rota, será então pedido ao passport autenticar, pelo modo “Local” o request recebido, caso o retorno seja de sucesso, ou seja, foi encontrado um usuário, este será redirecionado para a página main da aplicação, no nosso caso é http://localhost/, caso não encontre, será redirecionado para a mesma tela de login.

```
  app.post('/login',
  passport.authenticate('local', {

successRedirect:'/',
      failureRedirect:'/login'}) );
```

Listagem 7 – Autenticação Passport.js

Com isso podemos ver que a autenticação pelo Passport.js é muito simples, não requerindo muitos códigos para validar o usuário informado no banco. O que facilita durante o tempo de desenvolvimento de uma aplicação, pois o tempo de desenvolvimento de códigos de objetos que acessem o banco de dados e o retorno até o usuário é uma das partes trabalhosas durante o desenvolvimento de uma aplicação web.

4 Proxy

Um proxy, também conhecido como servidor proxy é basicamente um computador que intercepta requests processados, ou seja, quando você está conectado à um proxy, seu computador redireciona todos os seus requests de acesso para o servidor proxy que então processa de acordo com suas restrições e retorna algo para o computador que originou o request. Dessa forma, se conecta à um servidor de maneira indireta.

Proxys são muito usados para restringir acesso à certas páginas, filtrar conteúdo e navegar anonimamente na internet. Para que se possa navegar de forma anônima, ao se conectar a um proxy, ao invés de aparecer o seu IP como endereço, aparecerá o endereço do proxy.

Um dos fatores importantes de ressaltar sobre proxies é que eles fornecerão segurança e anonimidade, já que ele é quem controla todo o tráfico de acesso, ou seja, o proxy sabe de todas as tentativas de acesso - contanto que a conexão não seja SSL- e do seu IP verdadeiro.

Em contrapartida, o uso de proxies costuma reduzir a velocidade de conexão, visto que todos os requests passarão para outro computador processar antes de ir buscar ou não o conteúdo desejado.

4.1 NGINX

Segundo seu site oficial, certa de 12.18% dos sites ativos no mundo utilizam NGINX, totalizando cerca de 22.2 Milhões de sites.

Diferente dos servidores tradicionais, Nginx não se baseia em threads para lidar com requests, mas ao invés disso utilizam uma arquitetura assíncrona baseada em eventos, utilizando menos memória.

Com o aumento do acesso à internet global, os servidores tiveram que se adaptar para atendender milhares de acessos simultaneamente. Eis que surge o problema internacionalmente conhecido como C10K, onde um servidor teria que atender a 10 mil conexões ao mesmo tempo. O Kernel do Unix foi originalmente desenvolvido para controlar sistemas para um network de telefonia, separando assim -por mais que não tivesse sido utilizado nessa forma nos servidores OS - o controle do dado, isolando assim o Kernel de lidar com pacotes, gerenciamento de memória e processador, o que foi dado como responsabilidade as aplicações, assim qualquer dado pesado é dado para a aplicação, e o controle é feito pelo Linux.

Com o Apache, quão mais conexões forem solicitadas ao servidor, pior será sua performance, pois seu Kernel utiliza thread para processos, o que dá trabalho ao Kernel descobrir qual thread é responsável pelo pacote recebido. Além do pool de conexões em single thread, onde cada pacote passaria por uma lista de sockets, causando assim um problema de escalabilidade.

Com o uso de threads, o total de espaço alocado na pilha de cada thread pode causar o servidor de ficar sem memória virtual. O Nginx sabendo disso não utiliza threads para lidar com requests, mas sim uma arquitetura assíncrona.

O Nginx é mais novo em comparação com o Apache, o que já ajudou pois os problemas de outros servidores já eram conhecidos, o que o tornou uma das opções para o problema C10K, onde escala com facilidade em um hardware não tão potente, onde ele encaminha dinamicamente os requests para outro software que o tratará melhor.

No Apache existem vários módulos de multiprocessamento que informam como os requests do cliente são tratados. Em um desses módulos, também conhecido como mpm_prefork, cada processo gerauma thread única que lida com o request, o que acaba utilizando muita RAM.

Já no módulo mpm_worker, cada processo pode gerenciar várias threads. Já o mpm_event funciona parecidamente com o mpm_worker, porém ele mantém vivas as conexões.

> ”Esse tipo de processamento de conexão permite ao Nginx escalar muito com recursos limitados, já que o servidor é single-threaded e os processos não são gerados a cada nova conexão, logo o uso de memória e CPU tendem a permanecer consistente, até mesmo durante picos de acessos.” *(ELLINGWOOD, 2015, s.p).*

Como podemos observar na citação acima, no Nginx, seu algorítmo foi baseado em acessos assíncronos, não bloqueantes e conexões orientados à evento, onde cada processo consegue gerenciar milhares de conexões. Todas as conexões são alocadas em um loop - esse loop constantemente verifica os processos - onde os eventos são processados de forma assíncrona e não bloqueantes, que uma vez que a conexão é fechada, ela é removida do loop, matendo-a desta forma consistente em qualquer momento da aplicação, até mesmo naquela com mais acessos.

4.2 node-http-proxy

node-http-proxy é uma biblioteca de proxy programável para protocolos de internet, que suporta protocolos de transmissão (TCP) e protocolos de comunicação como o HTTP (Protocolo de Transferência de Hipertexto) e HTTPS (Protocolo de Transferência de Hipertexto Seguro).

Com tal biblioteca, é possível criar dinamicamente um proxy de servidor com algumas opções customizáveis. Essas opções variam se o protocolo é HTTPS,

se o websocket passará por um proxu também, se encaminhará o request para outro proxy entre outros.

Dessa forma, ao criar um proxy, poderemos ficar ouvindo todas as requisições de um certo alvo, que seria um endereço na internet e validar seu acesso ao mesmo. Quando o sistema intercepta uma requisição que bate com as configurações informadas, o proxy realizará a lógica informada para trabalhar no request recebido.

```
var httpProxy = require('http-proxy');

var proxy = httpProxy.createProxyServer({});

http.createServer(function(req, res) {
  proxy.web(req, res, { target: 'http://localhost.com:8080' });
});

proxy.on('error', function (err, req, res) {
  res.writeHead(500, {
    'Content-Type': 'text/plain'
  });
  res.end('Erro.');
});

proxy.listen(8005);
```

Listagem 8 – Como criar um Proxy

Como podemos verificar na listagem 8 para criar um proxy precisa-se apenas informar o tipo do proxy, e o local que esse proxy estará escutando, quando uma requisição bate com a fornecida, ele transforma o request em dois objetos, *req* e *res*, estes para request e response por meio do proxy para o retorno para quem originou o request.

Além disso, ainda conforme a listagem 8, podemos observar que também é possível interceptar erros que possam ocorrer durante a execução do proxy, onde caso aconteça qualquer erro, será retornado ao cliente por meio do objeto *res* (response) um código de erro 500, que é um código de erro genérico do protocolo HTTP, além de enviar uma mensagem customizada de erro para o cliente fornecendo maiores informações sobre o problema encontrado.

Como podemos ver nos exemplos acima, eles são válidos apenas para o protocolo HTTP, não contemplando o HTTPS(SSL), que é um protocolo com

certificado de acesso, para contemplar também o protocoloco com certificado, ao invés de criar um servidor proxy com apenas o alvo HTTP, a criação do servidor proxy deverá ser parecido com a listagem 9.

```
httpProxy.createServer({
  ssl: {
    key: fs.readFileSync('valid-ssl-key.pem', 'utf8'),
    cert: fs.readFileSync('valid-ssl-cert.pem', 'utf8')
  },
  target: 'https://localhost:8080',
  secure: true
}).listen(8005);
```

Listagem 9 – Como criar um servidor Proxy

Conforme podemos ver na listagem 9, estamos informando que o alvo *localhost*, na porta *8080* e com protocolo *HTTPS* requer uma segurança SSL, onde será lida a chave e certificado delas com codificação em UTF-8, e apenas com os dados validos que qualquer tratamento no request adiante será realizado.

Além de dos dois protocolos informados acima, também é possível proteger com proxy um websocket, onde sua criação também é muito semelhante às anteriores, onde:

```
httpProxy.createServer({
  target: 'ws://localhost:8080,
  ws: true
}).listen(8080);
```

Listagem 10 – Proxy WebSocket

Conforme listagem 10 acima, todo protocolo TCP de canal de comunicação passará pelo proxy, caso sua origem seja a mesma informada no objeto *target*, e que o objeto *ws:true* informa que o servidor proxy atenderá websockets.

Podemos observar então que com o uso da biblioteca *node-http-proxy*, facilita para o desenvolvedor a interceptação do request pra que ele seja tratado da forma que o projeto requer, deixando toda a lógica e validação para o mais próximo possível dos conhecimentos do desenvolvedor.

5 Sistema Gerenciador de Aplicações Web

O sistema SiGAW (Sistema Gerenciador de Aplicações na Web), é um sistema desenvolvido para o gerenciamento de aplicações na nuvem, que facilita por meio de uma arquitetura desenvolvida, a possibilidade de utilização de uma única aplicação para vários usuários em qualquer lugar, sem que esses tenham que prover de dependências para que a aplicação seja executada.

5.1 Cloud Computing

Cloud Computing, ou computação na nuvem, é uma forma de gerenciar recursos na internet e distribuir os recursos para todos os usuários que forem usar. Durante anos, muitas empresas, ao longo que o número de funcionários crescia, tiveram que aumentar o número de recursos para oferecer ao funcionário um ambiente onde ele pudesse produzir. O problema é que esses recursos, que podem ser tanto hardware para processar o recurso quanto software custam, o que não é atrativo para executivos.

Para isso, a possibilidade de disponibilizar o recurso apenas uma vez na nuvem e assim distribuir para quem fosse de direito era algo que interessava a muitos e com o desenvolvimento em nuvem isso se tornou uma realidade. De acordo com o site oficial do CSC

> *[...] O Cloud Computing representa uma utilização mais eficiente dos recursos, o que, por sua vez, se traduz em economia de custos e de tempo. (CSC, s.d).*

Podemos observar então a citação acima, que com a computação na nuvem é possível disponibilizar apenas um recurso e esse se propagar para todos os

usuários na nuvem, tendo que apenas investir em um recurso, ao invés de vários.

Dessa forma, em um sistema em computação na nuvem, todo o processamento do recurso será feita em apenas um local, podendo ser um computador central ou um servidor, ao invés de todos os computados existentes dos usuários, diminuindo assim o requerimento de grandes hardwares e softwares para todos os usuários.

Para acessar o sistema em nuvem, o sistema deverá disponibilizar uma interface para o usuário, que no caso pode ser um navegador e toda a parte interna é gerenciada pelo sistema.

Atualmente a solução em nuvem é muito utilizada por várias empresas, sistemas e-mail web como o Gmail, Hotmail e Yahoo! utilizam computação na núvem, onde através da interface disponibilizada, o usuário recebe dados da nuvem.

Existem duas partes de uma computação nuvem: back end e front end, onde ambos são acessados pelo usuário através de uma rede. O front end disponibiliza uma interface para o usuário acessar os dados e o back end serve como a parte "nuvem" do sistema, que controla os dados armazenados, computadores e servidores entre outros.

5.2 SiGAW

No sistema, existe um catálogo de aplicações que possam ser utilizadas pelo usuário. Ao selecionar uma aplicação, essa será instanciada em um container Docker que possibilitará que ela seja replicada em um navegador qualquer.

Com a utilização do Docker, ele nos possibilita a virtualização a nível de aplicação, ou seja, apenas será implementado o que de fato é um requerimento para a execução com sucesso de uma aplicação, diferente dos virtualizadores

a nível de máquina que virtualizam um sistema completo, adicionando maiores dependências à aplicações.

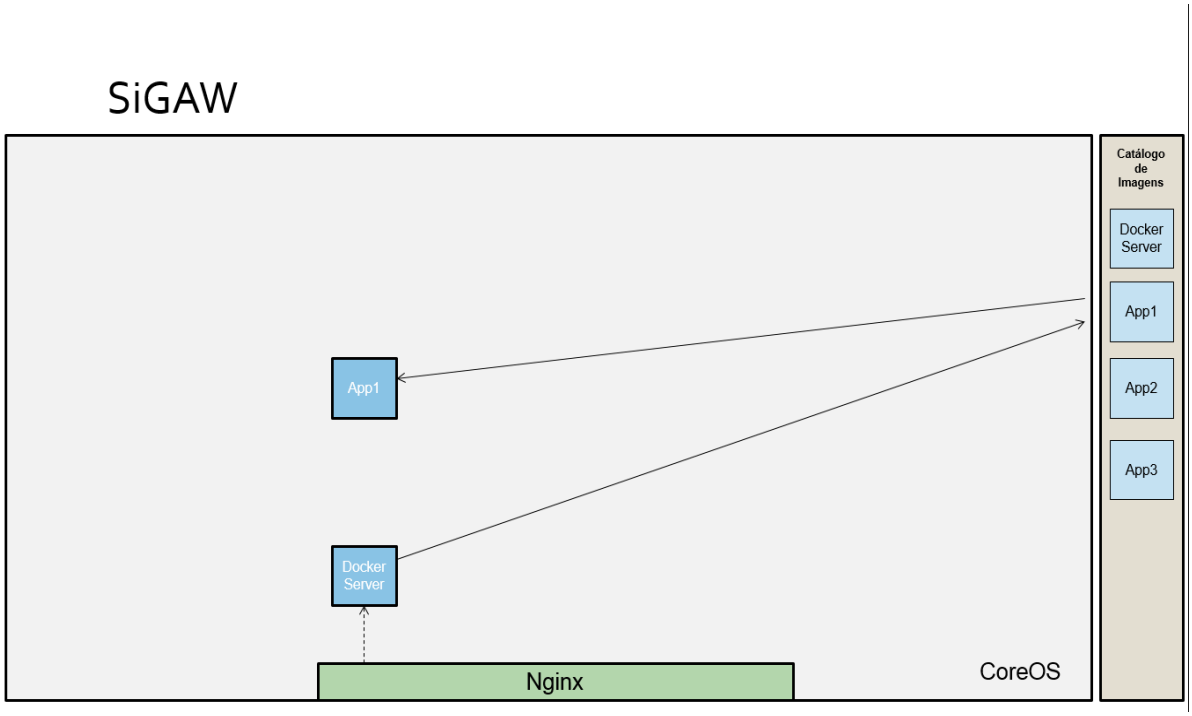
Um ponto interessante em qualquer aplicação na nuvem é sobre a disponibilidade do sistema. Uma das dificuldades é manter a aplicação sempre disponível para qualquer usuário a qualquer momento, para isso, na aplicação SiGAW foi desenvolvida uma lógica para que a aplicação esteja sempre online. Com isso foi desenvolvida uma forma de controlar a disponibilidade do sistema onde, caso por algum motivo um servidor Docker caia, através do etcd (Protocolo de Comunicação entre dois Dockers que funciona como pipeline), todas as tarefas que esse container estava realizando serão distribuídas para outros servidores estarem dando continuidade na tarefa utilizada.

No SiGAW, quando o proxy reconhece uma requisição de algum cliente, ele automaticamente encaminha para um servidor Docker receber a aplicação em um container. Vale ressaltar que a aplicação sempre terá pelo menos um servidor Docker em execução para trabalhar nos requests, para que sempre que chegar uma requisição nova de um cliente, o Nginx tenha para quem redirecionar.

5.2.1 Arquitetura

Para levantar um servidor Docker em uma solicitação, a própria aplicação valida se o(s) servidor(es) Docker na aplicação estão sobrecarregados, e caso esteja, a aplicação levantará outro servidor Docker e encaminhará o request para este novo servidor Docker tratar o request recebido.

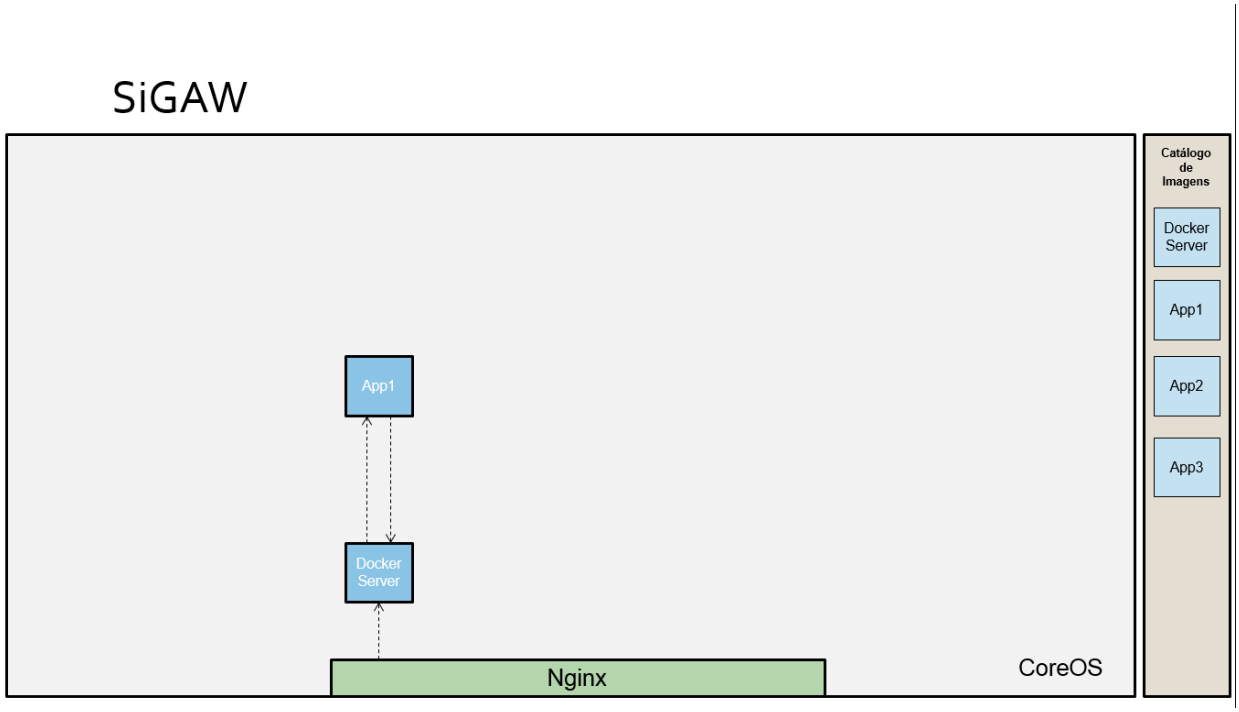
Figura 8: Arquitetura SiGAW



Fonte: (OS AUTORES, 2014)

De acordo com a Figura 8, quando a lógica implementada no Nginx reconhece o request, ele busca por um servidor Docker para encaminhar o request para ele, este que vai buscar a aplicação no catálogo de imagens e instanciar a nova aplicação - que chamaremos de App1 - que ficará disponível para o uso, porém o servidor Docker não tem como saber em qual local a aplicação foi gerada para poder retorná-la ao cliente, com isso, verificaremos a solução implementada na figura 9.

Figura 9: Solicitando uma aplicação.



Fonte: (OS AUTORES, 2014)

Como podemos ver conforme Figura 9, assim que a aplicação App1 é instanciada em um container Docker, ela mesma informa ao Servidor Docker sua localização, desta forma o Servidor Docker saberá onde está a aplicação e o que fazer com ela, e ficará em constante comunicação com a mesma, para saber quando ela será descartada ou não.

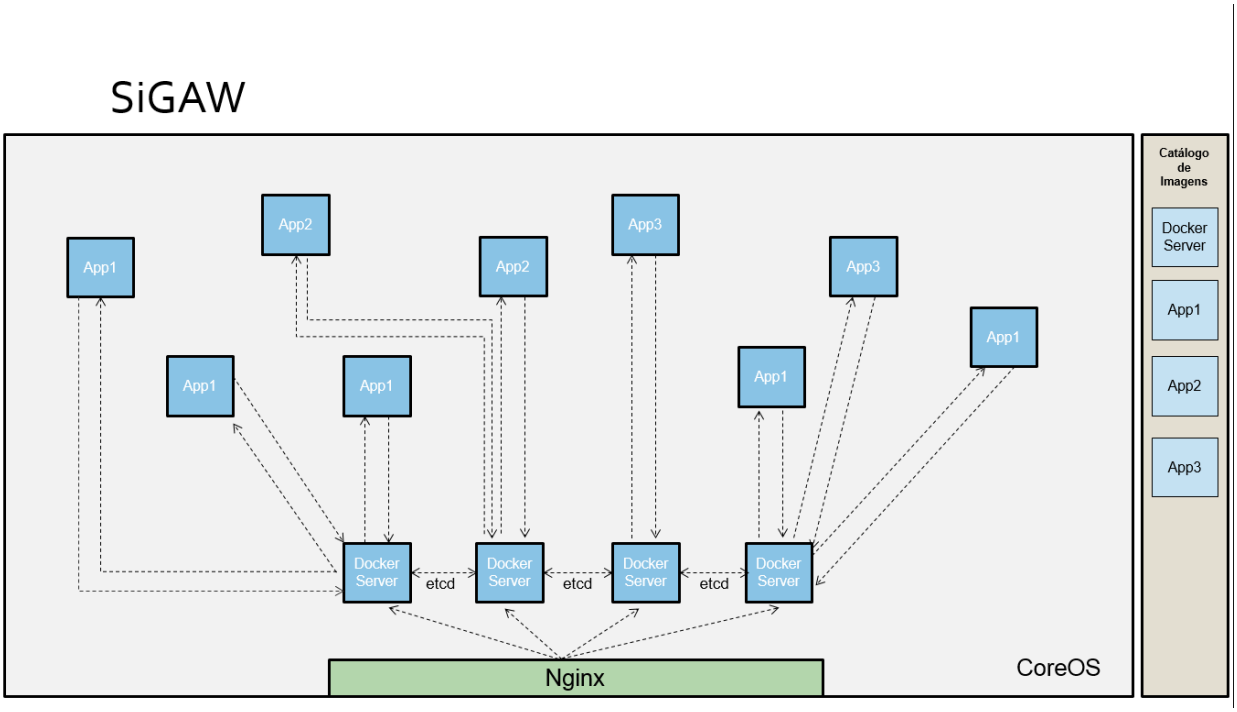
Como o problema de disponibilidade ainda é um problema que ser enfrentado na maioria dos sistemas em nuvem, uma solução para o problema foi implementada, onde a aplicação verifica se o servidor atual ficará ou está sobrecarregado, para repassar suas tarefas para outro(s) servidor(es), para isso todas as alterações deverão salvas no etcd, que é usado para a descoberta de serviços e valores de configuração de leitura e escrita. Este etcd conectará todos os servidores Docker para que eles se conheçam e saibam para quem encaminhar o request recebido.

Com o uso do etcd, todas as alterações realizadas serão salvas nele em nível de aplicação, permitindo assim ao usuário a realizar as alteraçoes em um

computador, e então dar continuidade no uso em outro computador, que verificará pelos valores escritos no etcd.

Logo após o servidor Docker conhecer a nova aplicação levantada, ele retornará para o usuário a aplicação App1, que novamente passará pelo proxy Nginx que então retornará a aplicação para o usuário em seu navegador.

Figura 10: Multi Solicitações de Aplicações.



Fonte: (OS AUTORES, 2014)

Como podemos verificar Figura 10, caso algum servidor por algum motivo caia, por o etcd conversar dois Dockers, cada Docker monitorarão outro, logo todos os servidores Docker acabarão sendo conhecidos e devidamente conversados, delegando todas as tarefas de um ao outro.

5.2.2 Implementação

O sistema é em sua maioria codificado utilizando JavaScript, para gerenciar as aplicações em nuvem para todos os usuários os usuários. Além disso, dentre

as aplicações utilizadas para teste do sistema, foi utilizada uma aplicação chamada Cloud9, que é uma IDE (ferramenta para programar e compilar).

Essa IDE foi colocada dentro de um container Docker assim que o servidor Docker reconhece através do request que o usuário deseja executar a IDE. Este servidor será usando para criar, destruir, gerenciar e proxy esse novo container levantado que contém o cloud9.

5.2.2.1 Levantando uma aplicação

Para levantar uma aplicação, utilizamos o seguinte código no arquivo dockerScript.js:

```
var nexpect = require('nexpect');

var username = process.argv[2];

nexpect.spawn("ssh core@172.17.8.101 docker rm -f " + username +
"Workspace:/" + username + "Workspace --name " + username + " --expose
4002 esdrasportillo/dockernode cloud9/bin/ cloud9.sh -l 0.0.0.0 -p
4002 -n " + username + " -w /" + username + "Workspace")

.expect(username)

.run(function (err) {

if (!err) {

console.log("Aplicação iniciada.");

} });
```

Listagem 11: Levantando uma aplicação em container Docker

Segundo Listagem 11, o código implementado em JavaScript descreve como levantar um novo container docker contendo uma aplicação cloud9 nela. Primeiramente, através de um objeto será levantada ela, que precisará ser devidamente configurada, para isso se é informado ao objeto o local na rede onde a aplicação deverá ser executada, o nome do usuário que o solicitou, pois o nome desse usuário será utilizado como chave para indicar qual container foi criado, para onde será enviado e quando será excluído, além do caminho para

a aplicação no Catálogo de Imagens, além de alguns outros comandos de configuração.

Ainda segundo o trecho da listagem 11, ao perceber que nenhum erro ocorreu ao se tentar iniciar a aplicação, a própria aplicação retorna uma mensagem de sucesso, onde será informada ao usuário o seu estado atual de que ela foi executada com sucesso.

5.2.2.2 Deletando uma aplicação

Para deletar uma aplicação, o seguinte código no arquivo deleteContainer.js foi utilizado:

```
var nexpect = require('nexpect');

var username = process.argv[2];

nexpect.spawn("ssh core@172.17.8.101 docker rm -f " + username )
.expect(username)

 .run(function (err) {

if (!err) {

console.log("Excluído");

 } });
```

Listagem 12: Deletando um container Docker

Conforme podemos perceber no código da Listagem 12, para deletar um container Docker através de uma função precisamos apenas identificar o Docker e sua localização. Para a identificação do Docker é necessário conhecer o nome do usuário que o solicitou, para que não seja excluída uma outra instância erroneamente. Sua localização é passa a ser um endereço na rede, pois ele já foi previamente entregue ao usuário pelo servidor Docker.

Caso não ocorra nenhum erro durante a exclusão do Docker, a aplicação retornará uma mensagem de que a instância Docker foi excluída com sucesso da aplicação.

5.2.2.3 Validando o Usuário

Para podemos retornar uma aplicação ao usuário, é necessário saber quem está utilizando e o que este mesmo usuário já realizou na aplicação. Para isso, esses dados são primeiramente validados de acordo com uma lógica onde, para poderem ter acesso à aplicação, deverão ser reconhecidos como usuários de uma conta Google.

Existem várias validações que poderiam ser implementadas, como por exemplo uma validação onde buscaríamos e salvaríamos em um banco de dados próprio local pelo usuário, porém, como o sistema será aberto para qualquer usuário, e não trabalharíamos com características a mais dele além de um simples nome como ID, foi optado por utilizar uma conta Google para fazer login do sistema conforme podemos verificar na seção de código abaixo no arquivo app/passport.js:

```
var GoogleStrategy = require('passport-google-oauth').OAuth2Strategy;

var configAuth = require('./auth');

module.exports = function(passport) { // usado para serializar o
usuário para a sessão

passport.serializeUser(function(user, done) {

done(null, user);

}); // usado para deserializar o usuário

passport.deserializeUser(function(obj, done) {

//User.findById(id, function(err, user) {

done(null, obj);

//});

}); // estratégia Google

passport.use(new GoogleStrategy({

clientID : configAuth.googleAuth.clientID,

clientSecret : configAuth.googleAuth.clientSecret,

callbackURL : configAuth.googleAuth.callbackURL,
```

```
},

function(accessToken, refreshToken, profile, done) {

process.nextTick(function () {

//profile.id = identifier;

return done(null, profile);

}); } )); };
```

Listagem 13: Criando uma estratégia de Login Google

Como podemos ver na seção de código acima, para validar criaremos uma nova estratégia de validação do tipo Google, passando três configurações necessárias para ela ser executada, que são clientID, clientSecret, callbackURL, que são respectivamente o email de usuário, sua senha e a URL de retorno.

Ao encontrar um usuário este será retornado serializado para a sessão, assim como também será deserializado para liberá-lo da sessão posteriormente.

5.2.2.4 O servidor

Neste arquivo nomeado de server.js será implementado a toda da lógica da aplicação, desde a validação do usuário até a criação e destruição de um container Docker. Nele, toda e qualquer requisição passará pelos seguintes passos:

1. O usuário conectará no Docker Server

2. O Servidor Docker redirecionará o usuário para a página principal

3. O usuário informa seus dados de login

4. O Servidor Docker faz uma chamada para o processo de Autorização Google

5.1 Se os dados informados são corretos, o usuário seleciona uma aplicação e o sistema começa a construir uma nova instância docker executando a aplicação selecionada usando Nexpect para SSH no CoreOS.

5.2 Caso os dados estejam incorretos, o sistema redirecionará o usuário para o passo 3.

6. O novo container Docker é criado e inicia a aplicação.

7. O novo container Docker envia um request para o Servidor Docker com seu IP e seu ID da sessão.

8. O Servidor Docker armazena as informações em uma hashtable.

9. Usando a ID da sessão atual, o Servidor Docker proxeia o usuário para a nova aplicação rodando em um container.

Para levantar um novo container Docker, precisa-se utilizar o arquivo dockerScript.js criado anteriormente, utilizamos a seguinte função:

```
ee.on("checked", function() {
child = exec('node ~/Project/dockerscript.js ' + userName,
function (error, stdout, stderr) {
 if (error !== null) {
console.log('Erro ao executar: ' + error);
} });  });
```

Listagem 14: Executando a função para levantar um Docker

Como podemos visualizar no código da Listagem 14, para executar o o arquivo dockerScript.js, através do arquivo server.js é necessário acioná-lo passado parâmetros que são esperados, que são o caminho de onde a função está mais o nme do usuário que o seleciona, para saber que o Docker que está sendo criado pertence a um certo usuário. Em caso de algum erro durante a execução da função, será exibido no console do navegador uma mensagem de erro com os detalhes do erro.

Para deletar uma instância em Docker, é necessário executar a função contida no arquivo deleteContainer.js, para acionália é executado o seguinte código:

```
child = exec('node ~/Project/deleteContainer.js ' + userName, function
(error, stdout, stderr) {

if (error !== null) {

console.log('Erro ao excluir: ' + error);

} ee.emit("checked"); });
```

Listagem 15: Executando função para deletar um container Docker

Conforme podemos verificar no código da Listagem 15 acima, seu código é muito parecido para subir uma aplicação, onde deveremos passar o local da função pra deletar mais o nome do Docker, para que não seja excluído por engano outro Docker.

Ainda segundo o código abaixo, caso a aplicação retorne alguma excessão durante a exclusão, que pode ser desde que o Docker já tenha sido excluído ou não existe um Docker com tal nome, será enviado para o console do navegador do usuário uma mensagem de erro com seus detalhes.

Quando a nova instância Docker é criada, o servidor Docker ainda não sabe onde este foi criado para poder devolvê-lo ao cliente, para que o servidor Docker conheça, é necessário que a nova instância informe onde ele está localizado, para isso, utilizamos o seguinte código:

```
app.post('/gotip', function(req, res){

var ipaddr = req.headers['my_ip'];

user = req.headers['user']; res.send("");

userssession[sessionID] = ipaddr;

console.log("user : " + usersip[user] + " session : " +
userssession[user]);

forwards[user](user);

delete forwards[user]; });

 app.listen(8080);
```

Listagem 16: Enviando o IP atual do container criado

Conforme o trecho da listagem 16, quando a nova instância é iniciada, ela envia um request do tipo POST para o servidor Docker, utilizando uma URL “/gotip” para passar o IP e o nome do usuário contidos no header do request.

Logo após, o endereço de IP é armazenado em uma hashtable, usando o ID da sessão do usuário como chave de identificação, encaminhando logo em seguida sua localização para o servidor.

Com as funções principais do sistema descritas acima, podemos ver de maneira bem isolada como cada função é executada e quais são suas necessidades para que elas sejam executadas com êxito na aplicação SiGAW.

6 Conclusão

A principal contribuição deste trabalho é desenvolvimento de um sistema sobre o cloud computing esclarecendo as principais vantagens e benefícios sobre a utilização de virtualização a nivel de aplicativos ao invés de se utilizar a virtualização a nivel de máquina, com âmbitos de apoiar a escolha de soluções em cloud para usuários num ponto de vista estratégico e econômico.

A contribuição desse estudo foi o desenvolvimento do SiGAW (Sistema de Gerenciamento de Aplicações Web) e com base nos resultados concluimos que é possivel criar um sistema que virtualiza aplicações para diferentes usuários que podem acessa-las de diferentes plataformas sem precisar de maiores dependencias para a execução do aplicativo. Com a lógica do sistema é possivel manter a aplicação sempre online, mesmo que um servidor caia, através da tecnologia Docker que distribuí a aplicação para outros servidores dando continuidade na tarefa utilizada.

Por usar CoreOS e Docker é possivel realizar um melhor controle das aplicações, além de torna-las mais leves para inicialização e restauração de estados sem que gere dependências.

Além disso, o fato de se usar Node.js para se manipular os containers Docker de forma assíncrona e não bloqueante permite que muitos usuários acessem o mesmo container sem que venha a causar deadlock e/ou grandes lentidões que sistemas em nuvem possam vir a enfrentar.

A sua arquitetura descentralizada permite uma abrangência global, capaz de integrar recursos pertencentes a diversos domínios sem grandes investimentos, características importantes para o trabalho colaborativo, a economia de recursos e na contribuição do meio-ambiente.

Este estudo teve como limitação a pesquisa bibliográfica devido a dificuldade na obtenção de obras nacionais a respeito do tema desta pesquisa, visto que muitas das tecnologias utlizadas no projeto foram lançadas entre 2013 e 2014.

Referências

BOND, Luke. **Getting Started with CoreOS and Docker using Vagrant**. 2014. Disponível em: www.lukebond.ghost.io/ acessado em 21/05/2014 às 18h.

CAMPOS, Augusto. **O que é Linux**. BR-Linux. Florianópolis, março de 2006. Disponível em: http://br-linux.org/faq-linux acessado em 14/04/2015 ás 17h.

CANNADAY, Brandon. **An Absolute Beginner's Guide to Node.js**. 2013. Disponível em:http://blog.modulus.io/absolute-beginners-guide-to-nodejs acessado em 12/04/2015 às 19h.

COREOS. **Using CoreOS**. 2014. Disponível em: https://coreos.com/ acessado em 21/05/2014 às 18h.

CSC, Tecnologia. **O que é Cloud Computing?** s.d. Disponível em: http://www.csc.com/pt/offerings/63346-o_que_%C3%A9_o_cloud_computing acessado em 01/03/2015 ás 14h.

ELLINGWOOD, Justin. **Apache vs Nginx:** Practical Considerations. 2015. Disponível em:https://www.digitalocean.com/community/tutorials/apache-vs-nginx-practical-considerationsacessado em 20/04/2015 às 16h.

FISHER, Paulo. **Hardware, Software e Sistemas Operacionai**, s.d. Disponível em: https://chasqueweb.ufrgs.br/~paul.fisher/apostilas/inform/sis_op.htm acessado em 22/03/2015 ás 19h.

GLOVER, Andrew. **Node.js para desenvolvedores Java**. 2011. Disponível em: www.ibm.com/developerworks/ acessado em 15/06/2014 às 8h.

GOOGLE. Chrome **V8:** Design Elements. 2015. Disponível em:https://developers.google.com/v8/design acessado em 20/04/2015 às 16h.

INTRODUÇÃO ao Sistema Operacional Unix, s.d. Disponivel em: http://www.cenapad.unicamp.br/servicos/treinamentos/tutorial_unix/unix_tutor.html#toc2 acessado em 22/03/2015 ás 18h.

LOWE, Scott. **Virtual machine vs Containers:** a matter of scope. 2014. Disponível em: www.networkcomputing.com/cloud-infrastructure/virtual-machines-vs-containers-a-matter-of-scope/a/d-id/1269190? acessado em 20/09/2014 ás 10h.

LYMAN, Peter; VARIAN, Hal R. **How much information?** Executive summary. 2003.

MARTINS, Ricardo. **Docker:** um linux container engine. 2014. Disponível em: www.ricardomartins.com.br/ acessado em 31/05/2014 às 21h.

POZZEBOM, Rafaela. **O que é Kernel?** 2015. Disponével em: http://www.oficinadanet.com.br/post/13858-o-que-e-kernel aceesado em 25/03/2015 ás 20h.

RITCHIE, Dennis M. and THOMPSON, Ken. ***The UNIX Timesharing System. Commun. of the ACM***, vol. 17, pp. 365-375, 1974.

TAURION, Cezar. **CLOUD COMPUTING:** Computação em Nuve**m** - Transformando o mundo da Tecnologia da Informação. Rio de Janeiro: Brasport, 2009. 226 p.

YEGULALP, Serdar. ***CoreOS uses Docker to put Linux on a diet.*** 2014. Disponível em: www.infoworld.com/ acessado em 21/05/2014 às 18h.