FACULDADE DE TECNOLOGIA DE AMERICANA

Curso Superior de Tecnologia em SEGURANÇA DA INFORMAÇÃO

Adriano da Silva

USO DE CONTAINERS PARA A OBTENÇÃO DE ALTA DISPONIBILIDADE E SEGURANÇA EM SISTEMAS DISTRIBUÍDOS

Uma abordagem com foco em pequenas e médias empresas

Americana, SP

2017

FACULDADE DE TECNOLOGIA DE AMERICANA

Curso Superior de Tecnologia em SEGURANÇA DA INFORMAÇÃO

Adriano da Silva

USO DE CONTAINERS PARA A OBTENÇÃO DE ALTA DISPONIBILIDADE E SEGURANÇA EM SISTEMAS DISTRIBUÍDOS

Uma abordagem com foco em pequenas e médias empresas

Trabalho de Conclusão de Curso desenvolvido em cumprimento à exigência curricular do Curso Superior de Tecnologia em SEGURANÇA DA INFORMAÇÃO, sob a orientação do Prof. Mestre Rossano Pablo Pinto.

Área de concentração: Segurança da Informação.

Americana, SP.

2017

FICHA CATALOGRÁFICA – Biblioteca Fatec Americana - CEETEPS
Dados Internacionais de Catalogação-na-fonte

S578u SILVA, Adriano da

Uso de containers para a obtenção de alta disponibilidade e segurança em sistemas distribuídos: uma abordagem com foco em pequenas e médias empresas. / Adriano da Silva. – Americana, 2017.

70f.

Monografia (Curso de Tecnologia em Segurança da Informação) - - Faculdade de Tecnologia de Americana – Centro Estadual de Educação Tecnológica Paula Souza

Orientador: Prof. Ms. Rossano Pablo Pinto

1 Segurança em sistemas de informação 2. Sistemas operacionais – containers 3; Cluster. I. PINTO, Rossano Pablo II. Centro Estadual de Educação Tecnológica Paula Souza – Faculdade de Tecnologia de Americana

CDU:681.518.5
681.3.066

Fatec Americana | Centro Paula Souza | GOVERNO DO ESTADO SÃO PAULO

Faculdade de Tecnologia de Americana

Adriano da Silva

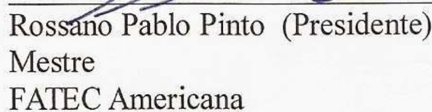
USO DE CONTAINERS PARA A OBTENÇÃO DE ALTA DISPONIBILIDADE E SEGURANÇA EM SISTEMAS DISTRIBUÍDOS

Uma abordagem com foco em pequenas e médias empresas

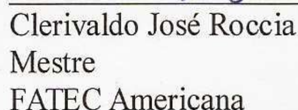
Trabalho de graduação apresentado como exigência parcial para obtenção do título de Tecnólogo em Segurança da Informação pelo Centro Paula Souza – FATEC Faculdade de Tecnologia de Americana.
Área de concentração: Segurança da Informação.

Americana, 11 de Dezembro de 2017.

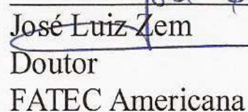
Banca Examinadora:



Rossano Pablo Pinto (Presidente)
Mestre
FATEC Americana



Clerivaldo José Roccia
Mestre
FATEC Americana



José Luiz Zem
Doutor
FATEC Americana

AGRADECIMENTOS

Em primeiro lugar agradeço aos meus pais Maria e José que me presentearam com a vida e tudo de bom que ela proporciona.

Agradeço aos amigos e professores que ao longo do curso compartilharam dos momentos mágicos de amizade e de aprendizado nas mais diversas pesquisas e debates que só enrriqueceram minha bagagem profissional e humana.

Agradeço aos funcionários da Faculdade, que sempre solícitos me ajudaram em muitas vezes quando precisei.

Agradeço de maneira geral à minha família e amigos que em momentos diversos aturaram minhas alterações de humor e rotina impostas pela demanda de trabalhos e estudos.

Agradeço ao meu filho Tomás e à minha esposa Daniela, por tolerarem em alguns momentos a minha ausência ou a falta da atenção devida.

À minha esposa que literalmente realizou a minha inscrição no vestibular para concorrer a uma vaga deste curso de graduação, me incentivou e, com muita paciência e compreensão, me ajudou ao longo deste percurso para que este momento se tornasse realidade, agradeço de maneira especial.

DEDICATÓRIA

Ao meu filho Tomás que me inspirou a progredir em busca de mudança de hábitos de vida, inclusive influenciando minha decisão de fazer um curso de graduação e à minha esposa Daniela que me incentivou muito a investir neste curso acadêmico e me apoiou em minhas fraquezas neste percurso.

RESUMO

Este trabalho investiga uma solução tecnológica para o fornecimento de serviços de processamento de dados tolerante à falhas baseada em *software* livre e financeiramente viável para implementação em micro, pequenas e médias empresas. A solução tecnológica deve ser capaz de reduzir o *downtime* dos serviços de informação nesse segmento de mercado, bem como a redução da quebra da integridade delas. Nesse sentido a tecnologia de containers de sistema operacional pode auxiliar na obtenção da alta disponibilidade dos sistemas de informação, pois consegue ser mais eficiente em termos de desempenho e economia de recursos quando comparado ao sistema tradicional de virtualização.

Palavras Chave: alta disponibilidade; cluster; container.

ABSTRACT

This work investigates a technological solution for the provision of fault-tolerant data processing services based on free and financially viable software for implementation in micro, small and medium enterprises. The technological solution should be able to reduce the downtime of information services in this market segment, as well as reduce the breakdown of their integrity. In this sense, the technology of operating system containers can help in obtaining the high availability of information systems, because it can be more efficient in terms of performance and resource savings when compared to the traditional virtualization system.

Keywords: high availability; cluster; container

SUMÁRIO

1 INTRODUÇÃO......3
1.2 Objetivos específicos......3
1.3 Motivação......4
1.4 Delimitação do escopo......6
1.5 Estrutura do TCC (trabalho)......6
2 CONCEITOS TEÓRICOS......7
2.1 Informação e sua importância......7
2.2 Relação entre a informação, comunicação e telecomunicação......8
2.2.1 Segurança da Informação......8
2.3 A tecnologia da informação......10
2.4 O processamento de dados......11
2.5 Sistemas operacionais e processos......12
2.6 Redes e sistemas distribuídos......16
2.7 Tolerância a falhas e alta disponibilidade......18
2.8 Computação em nuvem (cloud computing)......21
3 VIRTUALIZAÇÃO E ISOLAMENTO DE RECURSOS......24
3.1 Modelos de virtualização......25
3.2 Containers......31
4 CONTAINERS EM LINUX......36
4.1 chroot......38
4.3 Os cgroups......43
4.4 Capabilities para o usuário root......44
4.5 pivot_root......45
4.6 Mandatory Access Control (MAC)......45
4.7 Seccomp policies......46
4.8 Eliminando a influência automática de root......46
4.9 LXC – Projeto Linux Containers......47
4.10 Plataforma Docker......48
5 CENÁRIO: PROJETO DE ALTA DISPONIBILIDADE MONTADO EM CONTAINERS......51
6 CONSIDERAÇÕES FINAIS......60
REFERÊNCIAS......63

LISTA DE FIGURAS

Figura 1 – Pipeline de 5 estágios..14

Figura 2 – Processadores superescalares..15

Figura 3 – Categorias de nuvens...23

Figura 4 – Máquina virtual de processos...25

Figura 5 – *Os virtualization vs Hardware virtualization*..28

Figura 6 – *Hypervisor* tipo 1...29

Figura 7 – Hypervisor tipo 2..29

Figura 8 – Virtualização vs. Containers...33

Figura 9 – Container Linux...36

Figura 10 – Provisionamento de recursos no LXC...47

Figura 11 – LXC vs. Docker...49

Figura 12 – Docker para Windows..50

Figura 13 – Sistema do cenário...52

Figura 14 – Cenário físico..52

Figura 15 – Sistema de contingência..54

1 INTRODUÇÃO

A disponibilidade, a integridade e a confidencialidade são premissas fundamentais para a disciplina da segurança da informação (WOOD, 1984, p. 16), (ABNT, 2013, p. vi). Tendo isso em vista, identifica-se no objetivo deste trabalho (obter alta disponibilidade de sistemas de informação) a íntima ligação com o tema.

Isso porquê a alta disponibilidade de sistemas de informação cuida diretamente de garantir um tempo de disponibilidade quase contínuo desses sistemas que proveem o acesso à informação, ou seja, uma das condições básicas para a manutenção da segurança (TANENBAUM; STEEN, 2007, p. 195). Além disso, conforme Silberschatz, Galvin e Gagne (2010, p. 10–11) um sistema que está altamente disponível provavelmente estará tolerante à falhas, o que também assegura a integridade das informações, outro pilar da segurança, conforme já explicado.

A experiência profissional do autor deste trabalho, adquirida ao longo de muitos anos trabalhando com desenvolvimento de sistemas de informação e de infraestrutura de informática para pequenas e médias empresas, permitiu-o observar uma crescente necessidade dessas empresas por sistemas de informação automatizados e seguros ao mesmo tempo em que alegam dificuldades financeiras para investimento em tecnologias de segurança mais robustas devido a escassez de recursos, conforme também se confirma com Alvim (2015, p. 2–9).

1.1 Objetivo geral

O Objetivo geral do trabalho é propor uma solução tecnológica capaz de melhorar sensivelmente a disponibilidade da informação contida nos sistemas de processamento de dados de maneira viável ao mercado de micro, pequenas e médias empresas.

1.2 Objetivos específicos

Os objetivos específicos do trabalho podem ser enumerados da seguinte maneira:

1. Investigação de um sistema que contribua para que os serviços de informática estejam disponíveis 99,99% do tempo durante um ano e possam ser recuperados rapidamente em casos de desastres, tornando-se assim um sistema altamente disponível. Neste ponto vale ressaltar que itens como

redundância de hardware, link de internet ou sistema de fornecimento de energia elétrica contínua, embora contribuam para obtenção da alta disponibilidade, não são analisados neste trabalho por não fazerem parte do escopo.

2. A solução adotada deve aumentar a taxa de utilização dos equipamentos diminuindo o tempo de ociosidade dos mesmos, promovendo melhor aproveitamento e economia de recursos de sistema;
3. A solução adotada não deve incrementar o custo com licenças de *software*;
4. Finalmente, a solução adotada deve ter desempenho satisfatório;

Em outras palavras, um dos objetivos do trabalho é buscar uma solução em que a empresa possa ter segurança evitando elevar o consumo de recursos.

1.3 Motivação

É muito importante ressaltar que as MPE’s (micro e pequenas empresas) empregam 57,2% da mão de obra economicamente ativa no Brasil e são responsáveis por 20% do PIB, tendo como características a presença de familiares próximos aos cargos de gestão, decisões centralizadas e tomadas com base no conhecimento empírico do gestor, além da pequena quantidade de recursos financeiros disponível para investimento (ALVIM, 2015, p. 2–4). A despeito da limitação para investimentos supra mencionada, observa-se a enorme importância desse segmento empresarial para a geração de empregos e para toda a economia do país, pois trata-se de um mercado volumoso e com grande potencial para, de alguma forma, ser explorado.

Segundo Prates e Ospina (2004, p. 10), a informatização das MPE’s no Brasil tem tido um grande crescimento nos últimos anos. Afirmam ainda que um crescimento entre 30% e 80% tem sido registrado neste segmento dependendo da localização e da natureza do negócio. Entretanto, Rafael (2014) ressalta que neste segmento a maioria das empresas não valoriza a TI como um recurso estratégico, utilizando apenas como um ponto de apoio às operações da organização, cabendo a empresa prestadora de serviços de TI a orientação e o convencimento da importância da TI para os negócios.

Segundo Sorima Neto (2017), o setor de segurança da informação no Brasil ainda é dominado por multinacionais que geralmente tem como foco o atendimento das grandes empresas, deixando a fatia do mercado composta por micro, pequenas

e médias empresas carente de atendimento. Sorima Neto (2017) segue afirmando que há grande demanda por este tipo de serviço e a mão de obra especializada disponível é insuficiente. Carvalho (2013) confirma a importância desse segmento de mercado no Brasil, ressaltando ainda a demanda reprimida por conta da carência de mão de obra qualificada e da atual falta de cultura em investimentos em segurança da informação. Boa parte das pequenas empresas sequer sabem como ou em que precisam investir. A ausência de leis que regulamente no Brasil a exigência de um padrão de governança para as empresas, ou ainda a sensação de impunidade aliada à falta de *compliance* das empresas em relação as normas já reguladas, contribui para que a área de segurança da informação seja geralmente vista pela empresa como um custo sem retorno.

Em outras palavras, a falta de normatização para que as empresas busquem essa padronização e a ausência de uma cultura da Segurança da Informação na sociedade, são fatores determinantes para que a segurança da informação seja vista como um investimento sem retorno direto ao negócio e geralmente só discutida em empresas de grande porte. Carvalho (2013) segue completando que embora a segurança da informação seja importante para empresas de qualquer tamanho, no caso das empresas de pequeno e médio porte, o investimento neste setor tende a ser proporcionalmente menor, sendo que apenas 8% das empresas que tem entre 50 e 1.000 usuários possuem um profissional da área de segurança da informação (CARVALHO, 2013).

Alvim (2015, p. 8–9) também concorda que a tecnologia da informação é vital para a sobrevivência das MPE's, pois é um instrumento capaz de melhorar a capacidade criativa e aumentar a competitividade. Entretanto, explica também que devido a limitações financeiras e ao fato das pesquisas de desenvolvimento de tecnologia da informação não estarem voltadas as necessidades das organizações menores, há uma dificuldade dessas empresas em aderirem às soluções de TI.

No mesmo sentido, Prates e Ospina (2004, p. 16) indicam que os quesitos de custo, tempo e qualidade precisam ser observados para o sucesso da implementação de novas tecnologias nos ambientes das MPE's.

Rafael (2014) vai além e ressalta ainda a necessidade de se considerar soluções alternativas como as tecnologias baseadas em *softwares* livres como forma de redução do custo geral de implementação de serviços de TI.

As pesquisas realizadas neste trabalho consideraram estes quesitos.

1.4 Delimitação do escopo

A tecnologia escolhida durante a pesquisa da ferramenta para a realização da solução é baseada em *software* livre e derivada dos sistemas de virtualização, uma vez que o objetivo é atender aos requisitos de redundância de serviços de informação com redução do consumo geral de recursos.

Optou-se pelo sistema de container de sistemas operacionais por ser este um modelo que dispensa toda a camada de virtualização de *hardware*, bem como a camada *hypervisor*, dando maior desempenho e redução do consumo de recursos principalmente quando o objetivo é a redundância de muitos serviços que se auto-monitoram, ainda com a vantagem de ter agilidade na inicialização destes serviços.

Todos os termos técnicos apresentados nesta definição de escopo serão apresentados ao longo do trabalho, conforme estrutura apresentada na seção 1.5.

1.5 Estrutura do TCC (trabalho)

Este capítulo 1 trata da introdução do trabalho, sintetizando os objetivos de maneira geral e específica, além de relatar o que motivou a sua realização e delimitar o escopo. Por fim, explana a estrutura de todo o trabalho. O capítulo 2 aborda os conceitos teóricos sobre temas utilizados no trabalho de maneira geral. O capítulo 3 aborda a virtualização e o isolamento de recursos, um conceito teórico tão importante para a realização deste trabalho que mereceu um capítulo à parte. O capítulo 4 aborda a tecnologia de conteiners no sistema operacional Linux, um conceito teórico ainda mais específico, que está no cerne do desenvolvimento deste trabalho e por isso também recebeu seu próprio capítulo. O capítulo 5 trata do cenário da realização prática deste trabalho. O capítulo 6 trata das considerações finais.

2 CONCEITOS TEÓRICOS

Para uma boa compreensão do trabalho, este capítulo apresenta o conceito de informação e sua importância, a relação dela com a comunicação e com as telecomunicações, a segurança da informação e sua relação com este trabalho, a tecnologia da informação, incluindo uma sucinta história sobre a evolução tecnológica das telecomunicações e dos computadores, uma definição sobre o processamento de dados, processos, paralelismo, redes de computadores e sistemas distribuídos, tolerância à falhas, alta disponibilidade e computação em nuvem.

2.1 Informação e sua importância

Tudo aquilo que qualquer ser humano venha a conhecer ou a saber sobre a vida, pessoas, objetos, materiais, fenômenos, natureza, seres ou coisas é denominado informação. O conhecimento é o entendimento, a internalização e a incorporação da informação. Qualquer coisa que se saiba é uma informação incorporada, aprendida que pode ser exalada novamente como informação, conforme podemos experienciar. A esse respeito, descreve Ferreira (2010, p. 1158) que o termo “informação” pode significar simplesmente “conhecimento” ou ainda referir-se a “comunicação ou notícia trazida ao conhecimento de uma pessoa ou público”. Pode também referir-se aos “dados acerca de alguém ou algo”, além de instrução ou direção, bem como ao ato ou efeito de comunicar ou ainda participar.

Vieira (2009, p. 60) ensina que viver em comunidade demanda alto grau de troca de informações entre os indivíduos e desde os tempos mais remotos até o final da idade média no século XVII, a informação já era sentida como um bem de valor muito importante aos seres humanos. Importância esta que se intensificou durante o passar do tempo. Nas guerras, os generais passaram a utilizar a informação de maneira estratégica, assim como utilizavam técnicas para camuflar e protegê-la, ou ainda para roubá-la, como quem o faz para com uma preciosidade, um bem não apenas útil, mas valioso, raro e estratégico. Foi justamente neste longo período que a humanidade viu nascer a espionagem como forma de roubar as informações e também a criptografia como técnica para protegê-la (ocultá-la).

Com o passar do tempo, a humanidade percebeu que a detenção de informações ou conhecimentos verdadeiros e importantes é que define quem

realmente tem o poder nas mãos, dando abertura para o que conhecemos hoje como a era da informação.

2.2 Relação entre a informação, comunicação e telecomunicação.

Quando se fala a respeito da transmissão das informações, o uso do termo “comunicação” é talvez o mais apropriado, pois refere-se ao ato ou ao efeito de transmitir a informação ou conhecimento (comunicar, tornar comum), bem como refere-se também à transmissão de informações através de códigos, mensagens (faladas ou escritas) ou ainda outros sinais (FERREIRA, 2010, p. 545 – 546).

Conforme Defleur e Ball-Rokeach (1992, p. 102), com o aumento da complexidade das sociedades, verificava-se a necessidade de meios seguros para que as pessoas pudessem se comunicar, principalmente à grandes distâncias, pois a voz humana já não tinha alcance suficiente para propagar a informação entre as pessoas, nem tampouco um mensageiro seria capaz de percorrer longos percursos de maneira satisfatória para levar a informação de um ponto a outro. Então, a busca por meios de comunicação mais eficazes sempre existiu (som de tambores, sinais de fumaça, tochas em cima de morros). A criatividade dos seres humanos, segundo os autores, proporcionou a evolução da tecnologia da comunicação ao longo dos séculos, sendo que muitas técnicas foram inventadas ainda antes do ser humano conhecer e dominar a eletricidade.

O processo de comunicação a longa distância que utiliza como meio os fios telegráficos, telefônicos, as ondas eletromagnéticas ou meios similares é conhecido como **telecomunicação** (FERREIRA, 2010, p. 545 – 546).

Hoje em dia a informação se propaga muito rápido por meio da telecomunicação que a leva também diretamente ao processamento e armazenamento em computadores e dispositivos computacionais, conforme disposto na seção 2.6. Desta maneira a segurança da informação também deve compreender técnicas que assegurem essas informações enquanto dispostas, armazenadas, transmitidas ou processadas nesses meios.

2.2.1 Segurança da Informação

À medida em que a comunicação evoluía, ficava mais evidente ao ser humano a grande importância da informação. Mas foi somente nos últimos dois

séculos que a informação passou a ter importância crucial, principalmente nos governos, empresas e organizações em geral:

> Acima de tudo, **o bem mais valioso de uma empresa pode não ser o produzido em sua linha de produção, mas as informações relacionadas com esse bem de consumo ou serviço**. É importante que os executivos em geral se conscientizem de que **todas as informações tem algum tipo de valor** para alguém e/ou para algo; o que ocorre é que ainda não se descobriu para quem ou para quê. […] Atualmente não há organização humana que não seja altamente dependente da tecnologia de informações financeiras relacionadas com valores seus e de seus clientes. A maior parte desses dados é de natureza sigilosa, por força de determinação legal ou por se tratar de informações de natureza pessoal ou que controlam ou demonstram a vida econômica dos clientes, que podem vir a sofrer danos caso sejam levadas a público […] Não importa o meio físico em que as informações residam, elas são de valor inestimável não só para a empresa que as gerou como também para seus concorrentes. Em último caso, mesmo que as informações não sejam sigilosas, na maioria das vezes elas estão relacionadas com atividades diárias da empresa e sem elas poderia haver dificuldades. […] Assim, ainda que as informações não sejam passíveis do mesmo tratamento fisco-contábil que os outros ativos, do ponto de vista do negócio elas são um ativo da empresa e, portanto, devem ser protegidas. Isso vale tanto para as informações como para seus meios de suporte, ou seja, para todo o ambiente de informações (CARUSO; STEFFEN, 1999, p. 21 – 23).

Para o ser humano e suas organizações, ficou cada vez mais claro a importância de assegurar que as informações estejam disponíveis ao acesso do proprietário (ou de quem deva acessá-las), mas também que a disponibilidade de uma informação qualquer não basta, pois é necessário que ela seja: Verdadeira, correta e que esteja íntegra. E além disso, percebe-se em muitos casos a necessidade de privar terceiros de ter acesso a este ativo precioso, destacando-se a necessidade da confidencialidade. Wood (1984, 16) afirma que a segurança deve se ocupar de proteger os sistemas de falhas que possam afetar a disponibilidade, integridade ou o sigilo das informações. Conforme afirma a ABNT (2013, p. vi), a adoção de um sistema de gestão de segurança da informação (SGSI) é uma decisão estratégica das organizações e deve ser influenciada pelas necessidade, objetivos e requisitos de segurança dessa organização. Está consignado naquela norma que um SGSI deve visar a preservação da **confidencialidade**, **integridade** e também da **disponibilidade** das informações importantes para a organização em quaisquer meios em que elas estejam.

Discorrendo mais uma vez sobre a importância estratégica da informação para os seres humanos observa-se ainda como a tecnologia moderna vem influenciando rápido na popularização do acesso à ela. Como em um espiral, ao

mesmo tempo essa mesma tecnologia também aumenta a percepção das pessoas em relação à importância da informação, que sempre estimulou o ser humano na busca por instrumentos que o auxiliasse a processá-las automaticamente no intuito de obter resultados mais precisos e rápidos, como está descrito na seção 2.3.

2.3 A tecnologia da informação.

Guimarães e Lages (1985, p. 1) relatam que historicamente alguns pesquisadores consideram o monumento paleolítico “Stonehenge” (2600 a.C. – 1700 a.C.) como sendo o primeiro computador construído pelo homem, pois este monumento construído com pedras gigantes dispostas em uma determinada posição, processava informações com base na luz do sol que passava por fendas entre as pedras criando efeitos de luz e sombras para prever eclipses lunares. Ou seja, aplicou-se naquela obra uma técnica em que haveria a entrada de informação referente ao posicionamento do sol, baseada na presença ou ausência de luzes que entravam através das fendas nas paredes de pedras em determinados pontos, e o resultado do processamento seria o conhecimento de quando haveriam eclipses lunares.

Segundo Tanenbaum (2007, p. 10), o primeiro computador eletrônico foi o COLOSSUS construído pelo governo britânico em 1943, mas guardado como segredo militar. Guimarães e Lages (1985, p. 2–16) explica que o primeiro computador eletromecânico de uso geral (MARK-I) entrou em funcionamento somente em 1944, enquanto em 1946 surgiu o primeiro computador eletrônico digital conhecido publicamente na época (ENIAC). Os autores ainda contam que em 1951 um computador (UNIVAC) foi produzido em escala comercial. Entretanto, o uso destes equipamentos era restrito, sendo que apenas governos e grandes universidades tinham acesso a eles.

Desde 1980 os computadores já haviam diminuído tanto de tamanho, peso, consumo de eletricidade e de preço que já era viável que uma pessoa pudesse comprar e possuir o próprio computador, em cada departamento de uma grande empresa, em pequenas empresas ou ainda comércios, em um pequeno escritório ou ainda na própria residência. Conhecidos como microcomputadores, eles demarcaram início ao segmento denominado de computadores de quarta geração. Nesta época a IBM também lançou seu modelo de microcomputador conhecido como IBM-PC que utilizava o processador 8088 da Intel. Foi ainda no decorrer da

década de 80 que surgiram os primeiros computadores portáteis do mercado, produzidos de tal forma (tão pequenos e leves) para que uma pessoa sozinha conseguisse transportá-los com facilidade durante o uso cotidiano (TANENBAUM, 2007, 10–14).

A computação ubíqua ou ainda computação pervasiva, refere-se aos computadores menores e embutidos em eletrodomésticos, relógios, cartões bancários e diversos dispositivos (TANENBAUM, 2007, p. 15).

2.4 O processamento de dados.

Tanenbaum (2007, p. 29) explica que “um computador digital consiste em um sistema interconectado de processadores, memórias e dispositivos de entrada/saída”.

Stallings (2010, p. 19 – 27) ensina que os dados introduzidos no computador são processados em uma unidade constituída eletronicamente denominada CPU (Central Process Unit ou unidade central de processamento) que também é conhecida como **processador**. Trata-se de uma parte física e eletrônica, a principal do computador, destinada ao processamento dos dados, bem como ao gerenciamento central de todo o computador, incluindo memória, discos, dispositivos controladores e dispositivos de entrada e saída de dados, como se fosse o “cérebro” que realiza cálculos e determina a lógica do fluxo dos dados no computador.

Manzano e Manzano (1998, p. 106 e 126) explicam que os sistemas de computadores incluem o *Hardware* que equivale a todo o equipamento (físico) do computador, como as peças (CPU e outros componentes eletrônicos), acessórios e o próprio computador em si, enquanto o *Software*, seriam as partes “flexíveis”, não palpáveis, ou seja, os dados (as informações) e os programas que instruem o processamento dos dados, todos armazenados no computador ou em meios de gravação que podem ser acessados pelo computador.

Stallings (2010, p. 56–57) também ensina que os programas nada mais são do que instruções (informações que dão a direção, instruem o computador) previamente descritas e ordenadas de forma lógica para conduzirem o computador a realizar as tarefas que levam ao processamento dos dados que nele entram e assim entregarem o resultado esperado.

As instruções de um programa são executadas por uma ou mais CPU(s) dentro de um computador.

2.5 Sistemas operacionais e processos

O sistema operacional é um programa que controla as atividades básicas do computador como um todo, intermediando a comunicação entre o *hardware* (máquina física) e os outros programas que atendem as necessidades gerais do usuário (GUIMARÃES; LAGES, 1985, p. 141–152). Os autores explicam que após entrar em funcionamento, normalmente o sistema operacional fica no controle do computador durante todo o tempo realizando seu papel de intermediador, alocando recursos do computador necessários para os programas resolverem os problemas a que foram propostos. Silberschatz e Galvin (2000, p. 23–25) explicam que a parte central do sistema operacional que sempre está no controle do computador é denominada ***kernel*** (**núcleo** do sistema operacional). Os autores completam dizendo que o sistema operacional é composto pelo *kernel* que é o programa central, e também de programas aplicativos. Os autores detalham que recursos como tempo de uso da CPU(s), memória primária, secundária, dispositivos de entrada e saída de dados e outros são gerenciados e alocados pelo *kernel*.

Segundo Tanenbaum (2003, p. 53–56), uma vez em funcionamento, o *kernel* também é responsável por procurar outros programas armazenados na memória secundária do computador quando é instigado pelo usuário a fazê-lo, ou quando é previamente programado para tal. Após encontrá-los, fazer a leitura copiando seus conteúdos para a memória principal e na sequência orientar os recursos computacionais a iniciarem as execuções de suas instruções. Esta tarefa é conhecida como carga do programa.

Nos sistemas de computadores modernos, vários programas são carregados para trabalharem simultaneamente em um modelo de processos, onde um processo é um programa que está na memória principal. Vários processos podem estar na memória simultaneamente compartilhando o tempo de cada recurso para a execução de suas instruções, cada qual em uma fração de tempo destinada a ele. O programa que orienta o computador para esse compartilhamento de tempo é o *kernel* do sistema operacional. Corroborando:

> O mecanismo de alocação da CPU para execução de processos constitui a base dos sistemas operacionais multiprogramados. A transferência do controle da CPU entre os processos, pelo sistema operacional, torna o

> computador geralmente mais produtivo [...] A **multiprogramação** tem como objetivo permitir que, a todo instante, haja algum processo sendo executado, para maximizar a utilização da CPU (SILBERSCHATZ; GALVIN, 2000, p.147).

Essa característica de existirem vários processos simultâneos em um computador é denominada de **multitarefa** ou **multiprogramação**, e é a característica da maioria dos sistemas operacionais modernos, sendo que o escalonamento dos processos é o procedimento executado pelo escalonador da CPU, uma parte do *kernel* do sistema operacional utilizada para organizar todo esse funcionamento (SILBERSCHATZ; GALVIN, 2000, p.117 – 124), (STALLINGS, 2010, p. 213 – 224).

Os processos podem estar em, basicamente, três estados: pronto, em execução e parado/bloqueado. Embora vários programas possam estar em execução simultaneamente na memória principal de um computador (existindo assim vários processos simultâneos), apenas uma instrução de um único processo é colocada para execução na CPU em cada ciclo de tempo da mesma, também chamado ciclo de *clock* ou ciclo do relógio:

> Informalmente, um processo é um programa em execução. A execução de um processo ocorre de maneira sequencial, ou seja, uma instrução após a outra. A qualquer instante, apenas uma instrução de um determinado processo é executada (SILBERSCHATZ; GALVIN, 2000, p. 114).

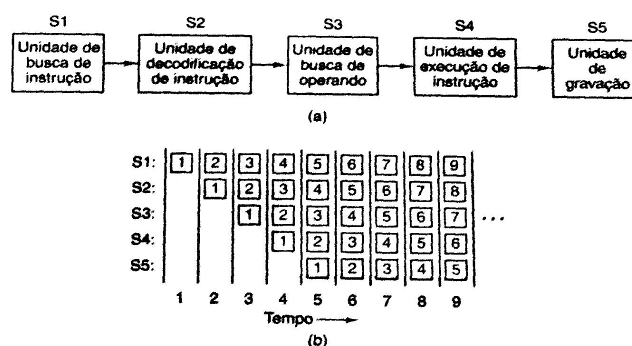
Cada instrução de um programa é executada dentro da CPU em vários ciclos de tempo, por um conjunto de procedimentos denominado **buscar-decodificar-executar**, que consiste em uma sequência de pequenas etapas, cada uma em seu tempo, sendo que estas se iniciam no primeiro ciclo de tempo tendo o processador (*hardware*) que buscar uma instrução na memória principal colocando-a em um registrador interno, posteriormente em um novo ciclo, ele deve guardar o endereço da próxima instrução em outro registrador, entender o tipo da instrução que será executada, verificar se a instrução precisa processar algum dado armazenado na memória principal, buscar este dado na memória colocando-o em outro registrador, executar a instrução em si e finalmente voltar à primeira etapa para buscar uma nova instrução a ser executada (TANENBAUM, 2007, 30–31). No mesmo sentido, outros autores ressaltam:

> Sempre que a CPU se torna ociosa, o sistema operacional deve selecionar um processo para execução, da fila de processos prontos. A seleção de processos é realizada por um programa chamado escalonador da CPU. Esse programa seleciona um processo dentre aqueles que estão na

> memória prontos para serem executados, e aloca a CPU para sua execução. [...] todos os processos na fila de processos prontos estão esperando para serem selecionados para execução (SILBERSCHATZ; GALVIN, 2000, p. 149 – 150).

Segundo Tanenbaum (2003, p. 16 – 17), no entanto, dizer que uma CPU é capaz de executar apenas uma instrução em cada ciclo, hoje em dia, é uma meia verdade. Isso porquê os projetistas de CPU, no intuito de obterem maior desempenho, desenvolveram novas tecnologias. Podemos iniciar falando da tecnologia *pipeline*, sobre a qual mais do que uma instrução pode estar sendo executada ou encaminhada simultaneamente. Nesta tecnologia, uma CPU é dividida em várias unidades independentes denominadas estágios, todas trabalhando em paralelo, cada uma responsável por uma tarefa relacionada à instrução. Por exemplo, para a tarefa buscar há uma unidade específica implementada. Para a tarefa decodificar há outro estágio independente, enquanto para a tarefa executar há um terceiro. É comum que processadores implementem *pipelines* com muitos estágios. A figura 1 ilustra o funcionamento de uma CPU com implementação de *pipeline* de cinco estágios:

Figura 1 – Pipeline de 5 estágios



Fonte: Tanenbaum (2007, p. 35)

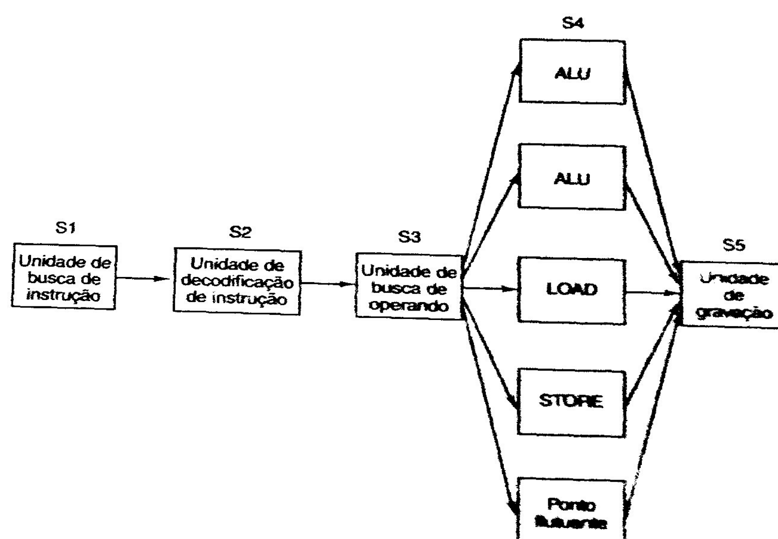
> Durante o ciclo de relógio 1, o estágio S1 está trabalhando na instrução 1, buscando-a na memória. Durante o ciclo 2, o estágio S2 decodifica a instrução 1, enquanto o estágio S1 busca a instrução 2. Durante o ciclo 3, o estágio S3 busca os operandos para a instrução 1, o estágio S2 decodifica a instrução2, e o estágio S1 busca a terceira instrução. Durante o ciclo 4, o estágio S4 executa a instrução 1, S3 busca operandos para a instrução 2, S2 decodifica a instrução 3 e S1 busca a instrução 4. Por fim, durante o ciclo 5, S5 escreve (grava) o resultado da instrução 1 de volta no registrador, enquanto os outros estagios trabalham nas instruções senguintes. (TANENBAUM, 2007, p. 35)

O processo de buscar instruções na memória sempre foi lento e desde 1959 a IBM já havia implementado um sistema que buscava antecipadamente as instruções

na memória e as armazenava em registradores internos, enquanto outra tarefa tarefa era executada pelo processador, em um estágio embrionário para as técnicas de *pipeline*.

Tanenbaum (2003, p. 16 – 17) explica que há também o caso dos processadores superescalares, onde dentro da mesma CPU existe uma unidade de execução para processamentos aritméticos de números inteiros, uma para processamentos lógicos e outra para processamentos aritméticos de números de ponto flutuante. E todas podem trabalhar em paralelo, sendo que a cada ciclo o processador estará executando uma instrução enquanto decodifica outra e busca uma terceira, ou ainda mais estágios são implementados. As instruções buscadas são acumuladas em um *buffer* de memória da CPU que as armazenam até que haja uma unidade de execução apropriada livre para executá-la. A figura 2 ilustra o funcionamento de um exemplo de CPU superescalar:

Figura 2 – Processadores superescalares



Fonte: Tanenbaum (2007, p. 36)

A iniciativa dos computadores superescalares é eficaz porquê o estágio S4 é mais lento que os outros e consome bem mais que um único ciclo de relógio da CPU para concluir sua tarefa.

Ainda no intuito de se obter incremento na rapidez do processamento dos dados, várias outras técnicas foram implementadas para que um computador pudesse executar mais do que uma única instrução simultaneamente em paralelo, sendo utilizado para isso mais do que uma CPU dentro do mesmo computador ou

ainda uma CPU com múltiplos processadores internos (conhecidos como núcleos), sem contar com outras técnicas como ***clusters***, que visam o trabalho cooperado entre vários computadores que se comunicam (STALLINGS, 2010, p. 514 – 574).

As técnicas em que vários computadores são utilizados para realizar serviços em conjunto, inclusive as redes de computadores, sistemas distribuídos e o processamento em paralelo dos *clusters* são abortadas nas seções 2.6 e 2.7.

2.6 Redes e sistemas distribuídos.

Desde os anos 50 até meados dos anos 80 (quando surgiram os computadores pessoais), a computação se apresentava em um modelo centralizado com computadores de grande porte conhecidos como *mainframes.* Estes eram muito caros, grandes e acessíveis apenas à empresas de grande porte, sendo que mesmo nesses casos a utilização deles se restringia à alta gerência. Nesse modelo os dados sempre eram processados nesse computador central e haviam os chamados “terminais burros”, que nada mais eram do que dispositivos de entrada (teclados) e saída (monitores) de dados espalhados em departamentos da empresa. Esses terminais não tinham capacidade de processamento de dados. Eles se comunicavam e eram ligados ao computador central (*mainframe*) responsável pelo processamento de tudo. Essa realidade mudou durante os anos 80 com o surgimento dos computadores pessoais (e dos microcomputadores em geral) que permitiram o surgimento de um modelo descentralizado de computação baseado em uma arquitetura denominada cliente-servidor (POLIZELLI *et al*, 2008, p. 117-118). Do que se entende da explicação de Tanenbaum (2007, p. 19 – 21), embora geralmente os computadores pessoais e servidores interconectados em uma rede tendam a ser menores em tamanho e capacidade quando comparados a um *mainframe*, cada um deles tem o seu sistema de processamento de dados autônomo e é comum que eles sejam numerosos em uma rede.

A tecnologia proporcionou a convergência entre os sistemas de processamento de dados (computadores) e as telecomunicações. Essa convergência evoluiu para uma verdadeira fusão entre as duas áreas, uma ligação técnica e íntima entre esses sistemas responsáveis por transmitir e processar os dados. Com isso, o conceito de computação centralizada também conhecido como conceito de “centro de computação” começou a ceder espaço para as redes de

computadores, onde as atividades de processamento de dados são realizadas em vários computadores interconectados (TANENBAUM, 2003, p. 1 – 3).

Sobre o surgimento dos microcomputadores poderosos em termos de processamento de dados, vale também ressaltar o dito por Stallings (2010, p. 29) sobre o surgimento das *workstations* (estações de trabalho) que são geralmente utilizadas para processamento de imagens e vídeos e dos Servidores, utilizados para entregar recursos ou serviços aos computadores clientes. Em suas redes massivas esses computadores vieram a substituir em grande parte os centros de computador *mainframe* do passado (STALLINGS, 2010, p. 29). Uma rede de computadores é um sistema onde podem ser interconectados desde microcomputadores e estações de trabalhos até computadores de grande porte (*mainframe*), passando também pelos minicomputadores. Nesse contexto, um computador é considerado um servidor quando disponibiliza um recurso ou serviço para ser utilizado por outro computador denominado cliente através das interconexões da rede. Existe uma confusão comum entre sistemas de rede e sistemas distribuídos. Para se caracterizar um sistema de rede, basta que haja uma comunicação entre dispositivos processados, mas o acesso aos recursos utilizados devem estar disponíveis ao usuário separadamente, cada qual em seu computador servidor e isso é perceptível ao usuário através da interface fornecida para acesso aos dados pelo computador cliente. O sistema que gerencia esse contexto é conhecido como sistema operacional de rede. De maneira diversa, em um sistema distribuído, embora os recursos estejam sendo providos através de uma rede e por diferentes servidores, ele é enxergado pelo computador cliente como se fosse um recurso local, ou seja, a existência de vários computadores provendo os serviços através da rede é transparente para o usuário que apenas percebe dispor dos recursos utilizados todos em seu computador local (SILBERSCHATZ; GALVIN, 2000, 503–504). Tanenbaum e Sten (2007, 194) ressaltam uma característica comum em sistemas distribuídos, relacionada com tolerância à falhas:

> Um aspecto característico de sistemas distribuídos que os distingue de sistemas de uma única máquina é a noção de falha parcial. Uma falha parcial pode acontecer quando um componente em um sistema distribuído falha. Essa falha pode afetar a operação adequada de outros componentes e, ao mesmo tempo, deixar outros totalmente ilesos. Ao contrário, uma falha em sistemas não distribuídos quase sempre é total, no sentido de que afeta todos os componentes e pode facilmente fazer o sistema inteiro cair (TANENBAUM; STEN, 2007, 194).

Para que um sistema distribuído seja confiável é necessário que ele esteja configurado para se prevenir de problemas, detectando-os e tomando providências necessárias para a recuperação de falhas objetivando a continuidade dos serviços, porquê também um sistema distribuído pode sofrer interrupções ou outros tipos de problemas causados por alguma anomalia no sistema, sendo que pode-se dizer que as mais comuns são mensagens perdidas durante a comunicação pela rede, defeitos em conexões ou em uma máquina (servidor) que participa do processo. (SILBERSCHATZ; GALVIN, 2000, p. 539–540). O tratamento de problemas com sistemas e a relação com os requisitos de disponibilidade e integridade das informações são tratados na seção 2.7.

2.7 Tolerância a falhas e alta disponibilidade.

Os sistemas de computadores processam dados das organizações, mantém acessíveis serviços essenciais relacionados à elas e, dada a importância deles para os negócios, espera-se que estejam sempre disponíveis e em bom funcionamento (WOOD, 1984, p. 13 – 17). Entretanto, alerta que um ataque ou uma falha pode ocorrer de maneira proposital ou acidental e inúmeras são as ameaças de ataques ou de falhas que podem perturbar o funcionamento adequado ou a disponibilidade dos sistemas. Exemplifica que uma ação humana, um desastre natural, um defeito no *software* ou ainda um defeito ou desgaste em um componente físico do sistema podem ser agentes causadores de problemas. Segue explicando que além da indisponibilidade do sistema ou das informações como um todo, o mal funcionamento de um sistema causado por uma falha ou por um ataque pode ainda causar a ruptura da integridade da informação pois o sistema não funcionaria conforme o esperado e assim a falha causaria a perda de parte da informação previamente armazenada ou ainda armazenaria uma informação errônea no lugar onde deveria estar uma informação correta.

Segundo Wood (1984, p.13 – 17), ainda por conta de falhas ou ataques, pode ocorrer também a quebra do sigilo (confidencialidade) da informação, quando a mesma é revelada a alguém que não estaria autorizado. A indisponibilidade ou mal funcionamento dos sistemas de informação computadorizados podem causar grandes aborrecimentos e prejuízos para as organizações. Para proteger os sistemas de que ataques ou falhas os afetem, deve-se tomar **medidas defensivas**, que por sua vez **elevam o custo dos sistemas**. Seguindo adiante, afirma que

dependendo das medidas tomadas pode-se obter um maior grau de proteção contra os ataque ou falhas, assim como também eleva-se o custo de implementação e que normalmente quanto mais medidas defensivas são tomadas, mais protegido o sistema estará e mais complexo, inconveniente e dispendioso ele se tornará. Entretanto, é preciso ter sempre em vista que o sistema jamais estará totalmente protegido de falhas ou ameaças que sempre poderão desencadear eventos danosos à informação. Desta feita os sistemas bem elaborados devem ter algum tratamento ou procedimento de contingência, ou seja, um plano para recuperação em casos de falhas. Apesar de que o investimento em medidas defensivas possa encarecer consideravelmente os sistemas, o custo com eventuais perdas de informações podem onerar ainda mais, de modo que se faz prudente contrabalancear investimentos em segurança conforme alertam Caruso e Steffen (1999, p. 49):

> As medidas de uma política de segurança devem ser, antes de mais nada, de cunho preventivo. Os eventuais riscos devem ser previstos e eliminados antes que se manifestem. Além disso, a prevenção costuma ser mais barata que a restauração dos danos provocados por falta de segurança.

Tanembaum e Steen (2007, p. 194) ensinam que a técnica fundamental para se manipular falhas é a **redundância** e o conceito de **tolerância à falhas** está ligado ao conceito de um sistema confiável, caracterizado quando atende aos seguintes requisitos:

1. Disponibilidade
2. Confiabilidade
3. Segurança
4. Capacidade de Manutenção

Neste sentido, asseveram os autores que o requisito de disponibilidade refere-se ao fato de um sistema estar pronto para seu uso em um determinado momento. Parecido porém distinto, ainda segundo os autores, o conceito de confiabilidade remete à característica de um sistema estar disponível durante todo um determinado período de tempo sem interrupção. Sendo assim, os autores observam que um sistema de **alta disponibilidade** é aquele que mais provavelmente estará apto a atender ao usuário em qualquer momento, enquanto o sistema de **alta confiabilidade** é aquele que provavelmente não sofrerá nenhuma interrupção no seu funcionamento pelo maior período de tempo.

> Se um sistema ficar fora do ar por um milissegundo a cada hora, terá uma disponibilidade de mais de 99,9999%, mas sua confiabilidade ainda será

> muito baixa. De modo semelhante, um sistema que nunca cai mas é desligado por duas semanas, todo mês de agosto, tem alta confiabilidade, mas somente 96% de disponibilidade. As duas não são a mesma coisa (TANENBAUM; STEEN, 2007, p. 195).

Segundo Lopes Filho (2008), um sistema de informação é classificado como sendo um sistema de **alta disponibilidade** quando ele se ocupa em **manter a disponibilidade** dos seus serviços (e consequentemente das informações pertinentes) para quem deva ter acesso, garantindo que os serviços estejam disponíveis (acessíveis) no mínimo durante 99,99% do tempo dentro do período de um ano. Um sistema que consiga garantir que as informações estejam disponíveis durante mais de 99,999% do tempo em um ano seria considerado um sistema de **disponibilidade contínua**, ou seja, um sistema que em termos de segurança na disponibilidade das informações é ainda superior aos classificados como de alta disponibilidade.

Continuando, Tanembaum e Steen (2007, p. 194) compreendem que o requisito de segurança é atingido quando um sistema tem a característica de falhar sem causar consequências desastrosas para o usuário. E alertam que é impossível criar um sistema completamente infalível, entretanto asseveram que um sistema que possa falhar sem afetar gravemente o usuário é considerado seguro. Por fim, explicam que a capacidade de manutenção é um requisito atingido pelo sistema que pode ser facilmente consertado após uma falha. E que, um sistema de alta capacidade de manutenção é um sistema que pode ser consertado muito rapidamente em caso de falha e consequentemente pode voltar a estar disponível rapidamente. Enfim, explicam que, caso um sistema de alta capacidade de manutenção tenha também a facilidade de detectar suas próprias falhas, assim como a capacidade de dar o início a manutenção desta falha e voltar a estar disponível de maneira automática, este provavelmente será também um sistema com alto grau de disponibilidade (TANENBAUM; STEEN, 2007, p. 195).

Os *clusters*, por exemplo, evitam a indisponibilidade do sistema, além de melhorar a performance do mesmo: “O cluster costuma ser usado para proporcionar serviço de alta disponibilidade, isto é, o serviço que continua mesmo se um ou mais sistemas do agrupamento falhar. Geralmente a alta disponibilidade é obtida através da inclusão de um nível de redundância no sistema” (SILBERSCHATZ; GALVIN; GAGNE, 2010, p. 10–11).

O ***clustering*** é um recurso de computação que provê um sistema distribuído escalável e estável, pois além de permitir que computadores sejam adicionados ao sistema dividindo o trabalho entre eles, o usuário não sofre prejuízos com relação a disponibilidade das informações caso um dos computadores venha a falhar, pois no *cluster*, mesmo no caso de falha de um equipamento, um outro computador sempre estará funcionando e provendo o serviço, proporcionando assim o que é conhecido como tolerância a falhas em um sistema de alta disponibilidade (STALLINGS, 2010, p. 532–538). Da mesma maneira, quando um computador que falhou reestabelece a comunicação com o *cluster*, ele deve voltar a assumir o seu papel de maneira transparente ao sistema cliente.

Massiglia e Marcus (2002, p. 302 – 372), bem como Thiel et al (2012), explicam que o conceito de ***failover***, utilizado em sistemas tolerantes à falhas, diz respeito ao processo que se dá quando um sistema falha e sua recuperação ocorre automaticamente, sem a percepção do usuário. Martins (2014, p. 34) afirma que o *failover* ocorre de maneira que "caso um nó do *cluster* vier a falhar, aplicações ou serviços passam a estar disponíveis em outro nó". Segundo Thiel et al (2012), o conceito de ***switchover***, ocorre quando o controle ou acesso do sistema é manualmente transferido de um servidor que pode ter falhado para outro. Neste mesmo contexto, conforme Massiglia e Marcus (2002, p. 302 – 372), ***failback*** (ou ***failover back***) é o conceito que se refere a recuperação do sistema, ou seja, quando o servidor que falhou é retornado ao seu funcionamento normal.

2.8 Computação em nuvem (*cloud computing*)

Segundo Aws (2017), computação em nuvem (ou *cloud computing*) é um tipo de estratégia onde um grande poder computacional é interligado para ser entregue em partes menores para o utilizador por meio da internet, sob demanda. Ou seja, conforme sua necessidade de utilização. Isso é, se o utilizador precisar de mais serviços ou mais poder de processamento, armazenamento ou volume de acesso, ele pode contratar e obter tais recursos de maneira muito fácil e rápida. Ou seja, o utilizador de onde estiver, acessa os serviços via internet, que está instalado dentro de um parque computacional (vários *datacenters*), com uma infraestrutura que oferece um sistema tolerante à falhas, e que deve estar espalhado geograficamente, redundante e interconectado por meio de uma rede de longa distância (internet). Além disso, esse sistema pode ter seus recursos escalados conforme a

necessidade. Este tipo de sistema pode ajudar o utilizador a economizar recursos financeiros por retirar a implantação e manutenção desse tipo de infraestrutura da responsabilidade dos utilizadores, passando-a para as empresas que fornecem o *cloud computing* (que obviamente cobram por isso).

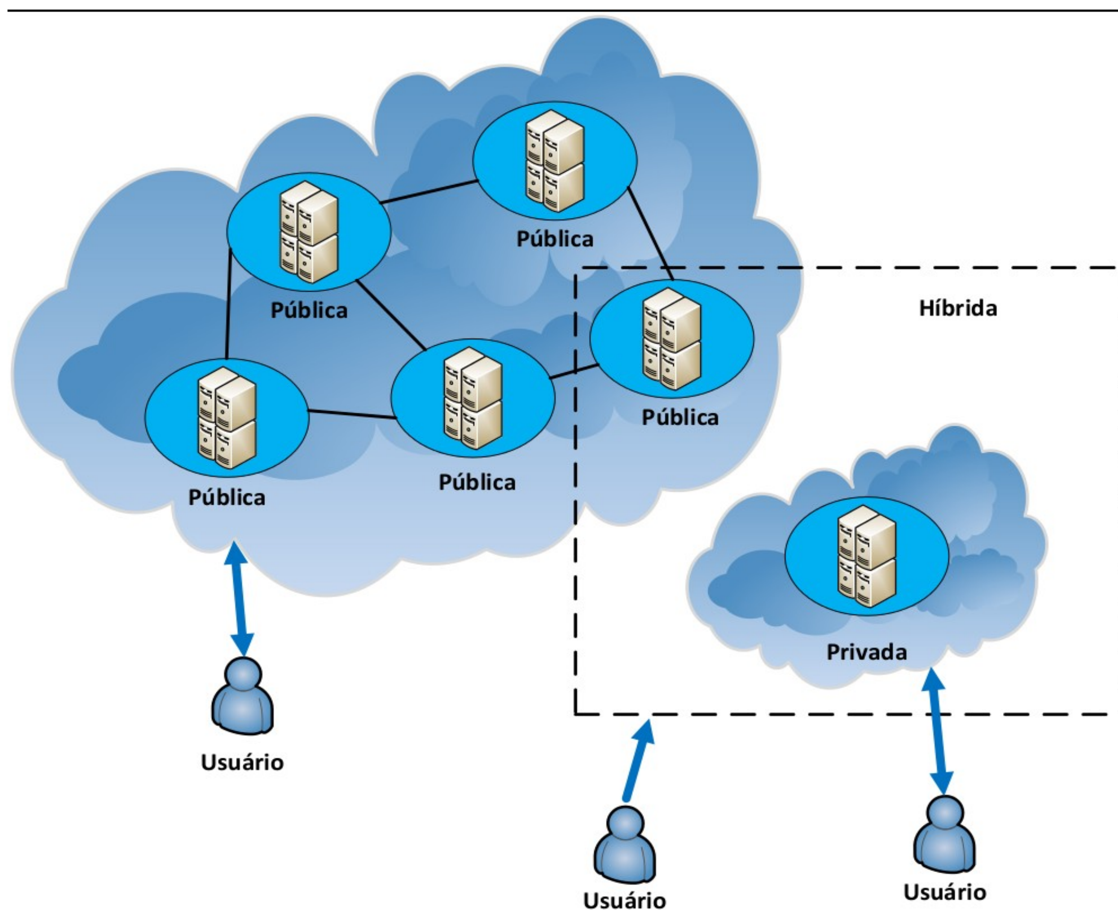
Segundo Martins (2014, p. 22), a questão fundamental da computação em nuvem é a utilização de muitos computadores interligados pela internet, sejam servidores ou até mesmo computadores de mesa, formando assim o poder computacional de um supercomputador. Dependendo das necessidades do usuário, pode-se utilizar diferentes arquiteturas para sistemas de nuvem:

- **Nuvem Privada** (*Private Cloud*): É um modelo onde a infraestrutura é provida para ser utilizada quase com exclusividade pelo cliente que a contrata, sendo que os serviços deste modelo não estão publicamente disponíveis, porém podem ser gerenciados por terceiros;
- **Nuvem Pública** (*Public Cloud*): é o modelo que está disponível publicamente, sendo que o cliente paga para utilizar sob demanda, oferecidas por grandes empresas ou organizações públicas que tem grande parque computacional disponível.
- **Nuvem Comunitária** (*Community Cloud*): É uma infraestrutura que pode ser administrada por terceiros ou pela comunidade que a utiliza, sendo que esta é composta por organizações que tem interesse em comum.
- **Nuvem Hybrida** (*Hybrid Cloud*): esta categoria é composta por duas ou mais nuvens de tipos diferentes (privadas, públicas ou comunitárias), interligadas por meios tecnológicos, devendo proporcionar a compatibilidade entre aplicações e seus respectivos dados. Estes meios de conexão podem ser padrão ou proprietários.

Martins (2014, p. 24) explica ainda que, em suma, as nuvens públicas oferecem ao usuário maior flexibilidade sob demanda, além de reduzir os custos com infraestrutura inicial, entretanto, não oferecem um controle mais apurado sobre questões de desempenho e segurança. No caso das nuvens privadas, apesar de menos flexíveis, os usuários tem maior controle sobre questões de desempenho e confiabilidade. As nuvens hibridas, todavia, tentam resolver o problema dos dois modelos combinando características de ambos, para que cada característica possa

ser explorada pelo usuário conforme a necessidade. Três das categorias de nuvem podem ser vistas na figura 3:

Figura 3 – Categorias de nuvens.



Fonte: Martins (2014, p. 23)

Segundo Martins (2014, p. 24), existe ainda a **nuvem privada virtual** ou *Private Cloud Virtual* (VPC), que utiliza a topologia de rede privada virtual (VPN). Neste modelo, as redes privadas do usuário são utilizadas em conexão com o sistema de nuvem com maior controle de segurança, uma vez que existe o túnel de criptografia entre as camadas de rede.

Aws (2017) explica que a computação em nuvem é uma tendência tecnológica em alta no mercado, e embora também seja um importante recurso computacional à disposição das empresas, a qual é citada várias vezes nesse trabalho, não será abordado em detalhes por não fazer parte do escopo.

3 VIRTUALIZAÇÃO E ISOLAMENTO DE RECURSOS

A virtualização é um processo onde uma máquina pode ser inteiramente emulada por um *software* sendo executado dentro de uma máquina física real (CACIATO, 2009, p. 4 – 5). Ou seja, trata-se de um programa que organiza dentro de um computador físico um ambiente computacional separado (virtualizado), onde nesse espaço as condições e funcionalidades de um equipamento completo de *hardware* são criadas para serem utilizadas isoladamente. Desta forma, um computador passa a ser capaz de executar mais de um sistema operacional simultaneamente, cada um com uma quantidade de memória RAM disponível reservada, tempo de processamento e tendo também seus processos em execução.

Caciato (2009, p. 4 – 5) afirma que esta técnica não é novidade, sendo que a ideia surgiu ainda em 1959 à partir do artigo *Time sharing processing in large fast computers* em uma conferência realizada em Nova Iorque, que trazia o conceito de multiprogramação em tempo compartilhado, com intuito de sugerir uma maneira de aproveitar melhor o tempo dos recursos de *hardware* de grandes computadores. Veras e Carissimi (2015, p. 1) explicam que deste trabalho surgiu o modelo CTSS (*Compatible Time Sharing System*) desenvolvido pelo *Massachusetts Institute of Technology* (MIT, [s.d]) e posteriormente utilizado por muitos fabricantes como referência, dentre eles a IBM.

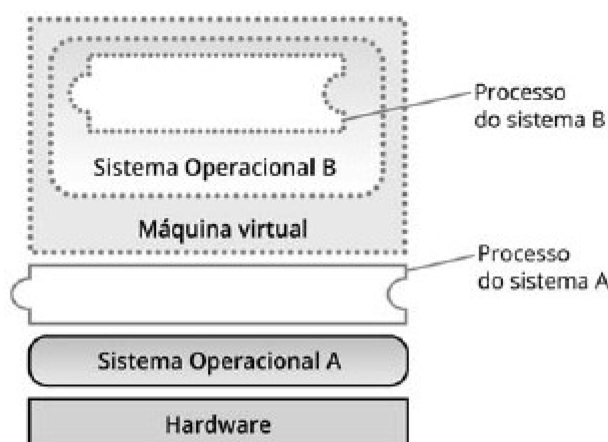
Caciato (2009, p. 4) afirma ainda que a primeira utilização prática de virtualização foi implementada pela IBM em seus mainframes no ano de 1965 e a introdução comercial nos computadores de arquitetura x86 passou a estar disponível desde o ano de 2001.

Assim, Tanenbaum (2003, p. 12–13) aduz, em alusão à afirmação de Charles Darwin a qual infere que a “Ontogenia recapitula a filogenia”, segundo a qual “o desenvolvimento de um embrião (ontogenia) repete (isto é relembra) a evolução das espécies (filogenia)”, que de maneira metafórica a evolução da tecnologia computacional moderna para a virtualização se assemelha a praticamente todos os ciclos evolutivos que ocorreram com outras tecnologias, envolvendo primeiramente os mainframes, em seguida os minicomputadores, microcomputadores, e por fim os celulares e outros pequenos computadores embutidos.

3.1 Modelos de virtualização

Para Veras e Carissimi (2015, p. 1–5) o conceito inicial adotado foi o de **máquina virtual de processo**. Nesse conceito, os binários de um processador podem ser executados em outro processador (podendo ser até mesmo de modelo ou arquitetura diferente) através de uma emulação que, segundo os autores é a “capacidade de imitar o comportamento externo de um sistema, sem preocupação com estados e propriedades internas a ele”. Afirmam ainda que neste modelo o sistema de virtualização substitui as instruções que um programa daria a um processador por instruções equivalentes de outro modelo de processador (o processador alvo). A ilustração deste modelo de virtualização, pode ser vista na figura 4:

Figura 4 – Máquina virtual de processos.



Fonte: Veras e Carissimi (2015, p. 2)

Por isso, os autores explicam que este modelo permite então, além da emulação de sistema operacional, também a emulação do processador em si, porém ao custo de diminuir o desempenho do sistema além de desperdiçar parte da capacidade do *hardware*. Como exemplo observa-se o projeto QEmu que, de acordo com o *site* oficial (QEMU, [s.d.]), é um *software* de código aberto emulador e virtualizador genérico capaz de emular máquinas de diferentes plataformas, mesmo que o *hardware* não tenha suporte de virtualização e, ainda assim, com desempenho satisfatório. Segundo informações do *site*, o *software* é capaz ainda de integrar-se com outros mecanismos de virtualização como o KVM (KIVITY *et al.*, 2007) e o Xen

(XEN PROJECT, 2013), quando pode conseguir em algumas situações desempenho próximo do nativo.

Os autores Veras e Carissimi (2015, p. 1–5) dizem que o conceito de virtualização pode ser categorizado em três níveis, que seriam:

- ***Hardware***: Onde o *software* de virtualização funciona em contato direto com o *hardware*. Exemplos de sistemas que oferecem esta categoria: VMware ESX (VMWARE, 2017) , Xen (XEN PROJECT, 2013) e Hyper-V (MICROSOFT, 2012);
- **Sistema operacional**: Virtualização em que o sistema operacional é separado em ambientes isolados logicamente, estabelecendo que tais ambientes sejam percebidos como máquinas separadas, dentro do mesmo SO. Exemplos de sistemas que oferecem esta categoria: Jails (RIONDATO, 2017), OpenVZ (PARALLELS, 2016), Solaris Zones (ORACLE, 2012), **Containers** (MCCABE; FRIIS, 2017) e (GRATTAFIORI, 2016), Linux-Vserver (VSERVER, 2013), Parallels Virtuozzo (PARALLELS, 2017), SandBox (GOLDBERG *et al*., 1996), KVM (KIVITY *et al*., 2007) e VirtualBox (ORACLE, 2017);
- **Linguagem de programação**: O *software* de virtualização funciona sobre o sistema operacional como uma aplicação, sendo que o mesmo é capaz de executar aplicações desenvolvidas sobre ele em linguagem específica de alto nível, própria do sistema. Exemplo: Máquina Virtual Java ou JVM (LINDHOLM *et al*., 2015).

Asseveram ainda os autores:

> As categorias acima possuem objetivos diferentes, mas buscam aspectos comuns: (i) oferecer compatibilidade de *software*; (ii) permitir o isolamento entre as máquinas virtuais, ou seja, um *software* em execução não deve ser afetado por outro; (iii) o encapsulamento, que possibilita a captura do estado da máquina virtual. A camada de virtualização deve ser projetada para não impactar o desempenho das aplicações (VERAS; CARISSIMI, 2015, p. 5).

Neste ponto vale ressaltar que, esta categorização do conceito de virtualização defendida pelos autores, no que se refere à virtualização em nível de sistema operacional, contempla diferentes técnicas de virtualização na mesma categoria sendo algumas baseadas em sistemas que compartilham o mesmo *kernel* com a máquina física como Jails, OpenVZ, Zones, Parallels Virtuozzo, Linux-Vserver, e outras baseadas em *hypervisor* como o caso do KVM que é parte

integrante do *kernel* Linux e do VirtualBox que é uma aplicação que roda sobre um *kernel* de sistema operacional.

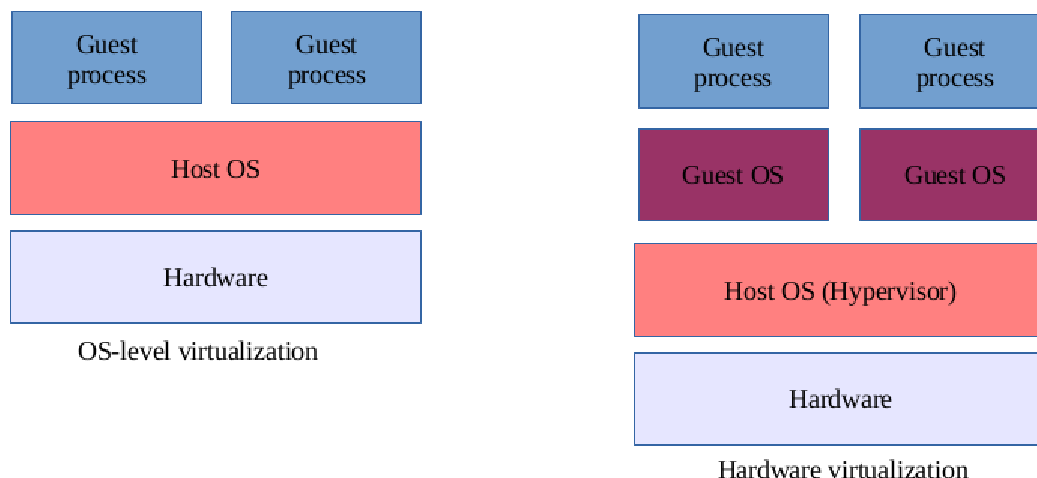
Grattafiori (2016, p. 7–10) faz algumas considerações importantes acerca da terminologia utilizada quando se trata do assunto de virtualização, indicando que é comum o emprego do termo "***host***" (com significado de hospedeiro, ou seja, aquele que hospeda, que recebe um visitante) para referir-se ao lado "real" do sistema, ou seja, o lado do *hardware* real (físico) junto do *software* tradicionalmente instalado diretamente nele (incluindo o sistema operacional principal e seus processos), jamais olhando para o lado do *hardware* e processos virtualizados. Também explica que é comum o emprego do termo "***guest***" (com significado de visitante, hóspede ou ainda hospedado), para referir-se ao sistema que está acomodado ou funcionando dentro do ambiente tido como virtualizado, incluindo o próprio *hardware* emulado por *software*, o *kernel* e os processos virtualizados, acomodados e funcionando dentro do sistema hospedeiro. Em outras palavras, isso significa que o *host* é o sistema real que hospeda os ambientes *guest* (hóspedes, visitantes) que nele estão virtualizados.

Grattafiori (2016, p. 7–10) também prefere destacar uma categorização entre os sistemas de virtualização feita de uma maneira diversa:

- O sistema que trabalha compartilhando o mesmo *kernel* entre o sistema hospedeiro e os sistemas hospedados é nomeado como ***OS-Virtualization***, ou seja, virtualização em nível de sistema operacional. Containers, Jails e Zones estão nesta categoria.
- Outra técnica é chamada de **paravirtualização**, onde o desempenho do ambiente virtualizado é muito parecido com o de um ambiente real. Porém para o uso desta técnica é necessário uma customização no *kernel* do sistema operacional hospedado (*guest*) para que as chamadas feitas por ele ao *hardware*, conhecidas como *hypercalls*, possam ser encaminhadas e recepcionadas diretamente pelo *hardware* físico real, sem serem intermediadas pelo *kernel* do sistema operacional hospedeiro (*host*).
- Além destas, existe a virtualização em **nível de *hardware***, técnica também conhecida como ***Full-Virtualization*** (virtualização completa). Nesses casos, um software chamado *hypervisor* cria e monitora ambientes isolados, onde em cada um deles simula o *hardware* da máquina virtual (chamado *hardware* virtual), podendo conter ainda, dentro desse ambiente isolado, todo um

sistema operacional sendo executado, incluindo seu próprio *kernel* e demais processos.

Figura 5 – *Os virtualization vs Hardware virtualization*



Fonte: Grattafiori (2016, p. 10)

Na figura 5 é possível observar, de um lado um tipo de virtualização em nível de sistema operacional onde o mesmo *kernel* de sistema é compartilhado entre todos os ambientes virtualizados, de outro um tipo onde há o *hypervisor* sendo executado sobre o hardware físico, ou seja, onde existe uma virtualização completa e sendo assim, há uma estrutura de *hardware* sendo virtualizada para cada máquina virtual além de sistemas operacionais hospedados (*guest*) cada um com seu próprio *kernel* isolado em cada máquina virtual.

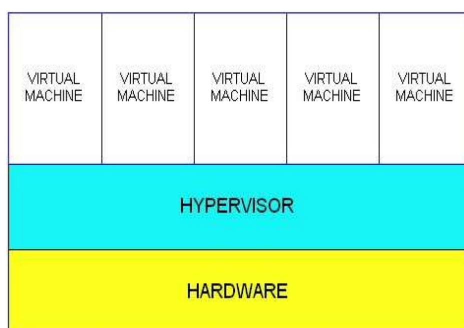
Para o autor, os sistemas de virtualização completa ainda são divididos em dois subgrupos menores, com importantes diferenças:

O sistema onde o *hypervisor* é o próprio sistema operacional hospedeiro, executado diretamente em contato com o *hardware* controlando-o sem nenhuma intermediação, chamado de *hypervisor type one* (**tipo um**). Classificados neste tipo os *softwares* temos Xen, VMWare ESXi (VMWARE, 2017) e o KVM.

E o sistema onde o *hypervisor* é executado sobre um *kernel* comum de sistema operacional hospedeiro que intermedeia a comunicação com o *hardware*. Este sistema é chamado de *hypervisor type two* (**tipo dois**). Aqui temos os *softwares* VirtualBox, VMware Workstation (VMWARE, 2011), e o QEMU.

Caciato (2009, p. 5), também ressalta a diferença entre os sistemas de virtualização, onde de um lado destaca os sistemas em que existe um *software hypervisor* que é executado diretamente no *hardware* daqueles onde o *hypervisor* é executado sobre um sistema operacional hospedeiro. Ele também ressalta que em ambos os casos existe um *software hypervisor* em execução, ambiente onde é simulada, hospedada, executada e controlada cada máquina virtual com toda sua estrutura interna:

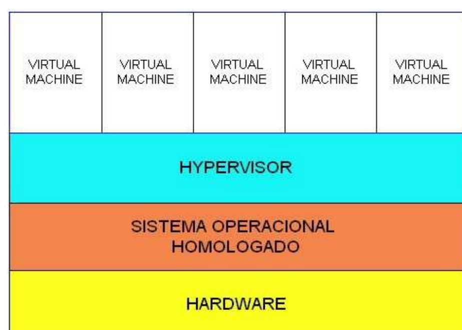
Figura 6 – *Hypervisor* tipo 1



Fonte: Caciato (2009, p. 5)

Como é possível verificar na figura 6, em um sistema onde o *hypervisor* é executado diretamente em contato com o *hardware* físico (*hypervisor* tipo 1), ele intermedeia toda a comunicação entre as máquinas virtuais que hospeda e a máquina física. De outro lado, um *hypervisor* desenvolvido para ser executado sobre um *kernel* de sistema operacional hospedeiro homologado, tem uma camada adicional de *software* que atua entre o *hardware* e as máquinas virtuais:

Figura 7 – Hypervisor tipo 2



Fonte: Caciato (2009, p. 5)

Na figura 7, observa-se que as máquinas virtuais são gerenciadas por um *hypervisor* (tipo 2) onde sua estrutura trabalha sobre o sistema operacional

hospedeiro, que por sua vez é executado diretamente no *hardware* físico. Ou seja, a comunicação entre as máquinas virtuais e a máquina física é intermediada pelo *hypervisor*, mas também necessita do *kernel* do sistema operacional hospedeiro, sendo este uma camada de *software* extra adicionada à pilha.

Veras e Carissimi (2015, p. 6–11) destacam também a existência da técnica de **virtualização completa assistida por *hardware***. Os autores explicam que nesta modalidade o processador físico precisa ter uma característica específica. Tal característica seria um conjunto extendido de instruções internas que permite que o *hypervisor* indique diferentes níveis de privilégio (uma espécie de prioridade) para as instruções. Assim, as chamadas ao *hardware* virtual realizadas pelo sistema operacional hospedado são interceptadas pelo *hypervisor* que as envia diretamente ao *hardware* da máquina física com um nível de privilégio mais alto que o normal, sendo assim executadas com grande rapidez.

Veras e Carissimi (2015, p. 6–11) dão o exemplo de dois modelos de implementação dessa técnica na arquitetura x86 de 64 bits: O “Intel VT” (INTEL, [s.d.]) e o “AMD-V” (AMD, 2017). Seguem ainda afirmando que esta técnica, apesar de exigir processadores específicos, permite uma virtualização em nível de *hardware* com desempenho muito próximo ao da virtualização com uso da ténica de paravirtualização, porém sem o ônus de ter que modificar o sistema operacional interno da máquina virtual, procedimento necessário quando se realiza as *hypercalls*.

Segundo KVM ([s.d.]), a partir da versão 2.6.20 do kernel Linux, a solução KVM (um software de código aberto) está disponível e implementa a técnica de virtualização completa para a arquitetura x86 com as extensões Intel VT e AMD-V, através do carregamento do módulo kvm.ko do *kernel* (trabalha em conjunto com os módulos kvm-intel.ko ou kvm-amd.ko, conforme o processador utilizado. No KVM pode-se utilizar várias máquinas virtuais sem a necessidade de se modificar o sistema operacional interno e que seu componente que roda no espaço de usuário (espaço onde o processador é chaveado para o modo menos privilegiado que existe) está incluído no código fonte principal do QEmu a partir da versão 1.3.

Em suas contribuições para o conceito, Seo *et al* (2014, p. 106–107) afirmam que *hypervisor* é uma técnica onde se tem várias máquinas virtuais e as mesmas podem ter recursos como CPU, memória RAM e interfaces de rede, todos

virtualizados e independentes, sendo que tais sistemas compartilhariam os mesmos recursos físicos para funcionarem. Em contrapartida, explicam os autores, a virtualização em nível de sistema operacional tem a mesma finalidade mas utiliza outra abordagem. Nela, o *hardware* físico e o *kernel* do sistema operacional hospedeiro utilizado é o mesmo compartilhado entre todo o sistema e os ambientes virtuais.

Segundo Baroni (2007), temos ainda o UML (User Mode Linux) que é uma maneira de se "construir uma máquina virtual com significado de um computador fictício criado por *software*", um ambiente onde um novo *kernel* Linux é executado como sendo um processo dentro do que chama de espaço do usuário. Neste novo *kernel*, algumas características chamadas *uml* são ativadas e por isso toda a comunicação dele com o *hardware* é traduzida para uma comunicação agora realizada com o *kernel* hospedeiro, que estará dando o suporte. Segue demonstrando ainda, como é possível copiar todos os arquivos de um sistema operacional Linux para dentro de uma área que pode ser utilizada por este novo *kernel* criando todo um ambiente completo para o carregamento e execução de outros processos. Desta maneira, trata-se de mais um modelo de virtualização dos recursos no nível do sistema operacional onde um novo *kernel* é inicializado e executado como um processo sendo executado por outro.

3.2 Containers

O mecanismo conhecido como container pode ser considerado um tipo de virtualização, porém com características próprias, distintas dos sistemas mais tradicionais que virtualizam o *hardware*, conhecidos como *Full-Virtualization* (virtualização completa). Difere muito também do sistema de paravirtualização. Para Grattafiori (2016, p. 9) e Docker (2017, 2017b), trata-se de um sistema de virtualização em nível de sistema operacional, onde o mesmo *kernel* do sistema operacional hospedeiro (*host*) é compartilhado entre todos os ambientes virtualizados.

Segundo Redhat ([s.d.]), a ideia de containers surgiu em meados dos anos 2000, para particionar o sistema operacional FreeBSD (MOCK, 2017) em ambientes isolados chamados de Jails. O sistema era considerado seguro, entretanto falhas detectadas nas primeiras versões permitiram que processos burlassem a segurança quebrando o isolamento do sistema.

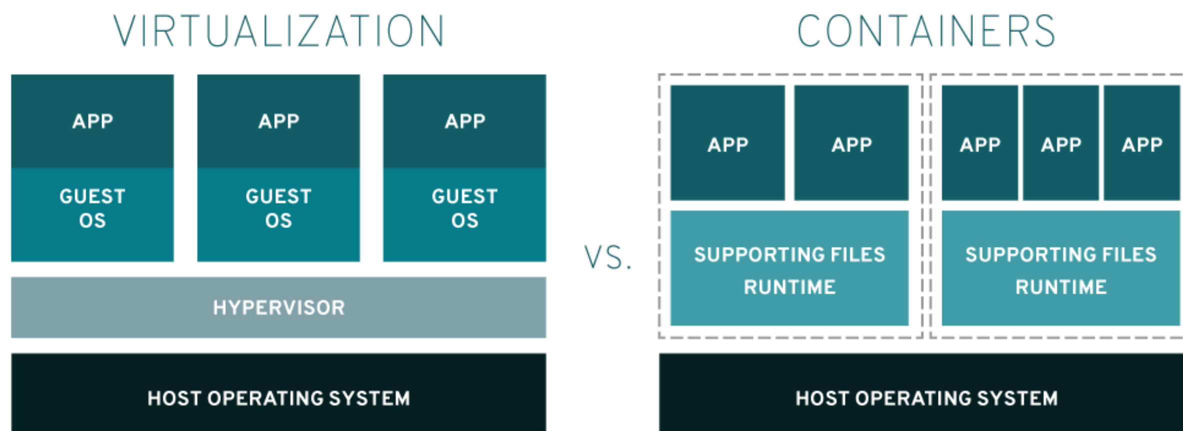
A Microsoft ([s.d.]), quando explica sobre container no sistema operacional Windows, afirma que é uma forma para se executar aplicativos em um ambiente isolado particular que abriga tudo o que tal aplicativo precisa, não tendo este acesso ou noção da existência ou inexistência de outros processos ou ainda de outros aplicativos que possam estar sendo executados fora do tal ambiente, ainda que externamente tudo esteja no mesmo sistema operacional. Segue explicando, de maneira metafórica, que este container é íntegro e pode ser levado para qualquer outro lugar (outro sistema de computador) que o hospede, e assim poderá executar as mesmas funções onde quer que esteja.

O Parallels (2016, 8 – 9) explica que a tecnologia de containers permite virtualizar servidores físicos na camada do *kernel* do sistema operacional que está em contato direto com o *hardware* físico, sendo que esta mesma camada é responsável por isolar os ambientes e os recursos de cada container fazendo este se sentir como se realmente fosse um servidor físico isolado.

A Redhat ([s.d.]) também enfatiza que embora container seja uma maneira de virtualizar um ambiente para que um aplicativo possa funcionar como se estivesse em uma máquina isolada, este difere dos modelos tradicionais de virtualização completa, também conhecida como virtualização em nível de *hardware*.

A Redhat ([s.d.]) explica que no modelo tradicional, além dos aplicativos, cada máquina virtual possui seu próprio sistema operacional com o *kernel* sendo executado isoladamente sobre uma estrutura de *hardware* fictício virtualizado pelo *hypervisor* e a partir de então, todas as máquinas virtuais são executadas sobre o sistema operacional hospedeiro. Em contrapartida, no modelo de container, o *kernel* do sistema operacional do hospedeiro é o único a ser executado e o tempo dele é compartilhado com cada container em execução. Na prática, explica a Redhat ([s.d.]), quando usamos container, as aplicações rodam imediatamente sobre o sistema operacional do hospedeiro, não havendo outras camadas ou *kernel* de sistemas operacionais hospedados (*guest)* intermediando a comunicação:

Figura 8 – Virtualização vs. Containers.



Fonte: Redhat ([s.d.])

Como pode ser visto na figura 8, em contraste com o modelo de virtualização completa, no modelo de containers não existe a camada *hypervisor*, onde todo o *hardware* seria virtualizado, assim como também não existem os sistemas operacionais visitantes (*guest*) dentro de cada máquina virtual, sendo que as aplicações e os arquivos de suporte (*runtime*) são executados diretamente pelo *kernel* do sistema operacional hospedeiro (*host*). Neste modelo, em comparação com o modelo de virtualização completa, existe um número menor de etapas durante a comunicação entre a aplicação executada dentro do ambiente virtualizado e o *hardware* físico que está logo abaixo do *kernel* do sistema operacional hospedeiro. Explica que por estas características, o modelo de containers se torna mais leve, portanto mais ágil para a realização da computação paralela de vários processos simultaneamente, promovendo economia de recursos quando comparado ao modelo de virtualização completa.

A Redhat ([s.d.]) ainda explica que um container pode ser concebido para executar apenas uma aplicação, ou ainda apenas parte de uma aplicação, como sendo um microsserviço. Segundo Fussell (2017) e Redhat (2016), microsserviço de uma arquitetura de microsserviços, refere-se a um pequeno serviço computacional autônomo pertencente a uma arquitetura de sistemas distribuídos baseada em pequenos serviços de baixo acoplamento, ou seja, eles podem trabalhar sozinhos não tendo dependência uns dos outros para seu funcionamento. Entretanto, o trabalho deles em conjunto pode oferecer uma solução completa, maior e demandada. Fussell (2017) explica que esta arquitetura implica em maior resiliência

aos sistemas, além de facilidades para o desenvolvimento, manutenção e escalabilidade. Este dinamismo, segundo Segundo Fussell (2017), é que pode receber contribuições do modelo de containers, sendo que um container é leve e dinâmico o suficiente para levantar ou dispensar serviços conforme a demanda em menor tempo e com menor consumo de recursos.

Para Docker (2017, 2017b), os containers também servem para isolar recursos ou aplicativos, inclusive para que os problemas que tal aplicativo venha a apresentar fiquem restritos apenas a aquele container que ele pertence, poupando outros containers ou o sistema da máquina principal.

Neste ponto vale destacar esta como sendo uma característica de contribuição direta do container para com os requisitos de **segurança da informação**, uma vez que desta forma o sistema de isolamento dos containers está contribuindo diretamente para a preservação de disponibilidade de funcionalidades do sistema e consequentemente da disponibilidade e integridade da informação.

Na abordagem explicada por Docker (2017, 2017b), o container de um aplicativo representa um conjunto leve de programas e arquivos que juntos viabilizam a execução de uma aplicação, independente da plataforma em que ele for levado, ou seja, independente do sistema operacional.

Docker (2017, 2017b) afirma que os containers são rápidos para inicializar além de ocupar espaço muito inferior quando comparado a máquinas virtuais tradicionais.

Docker (2017, 2017b) se diz líder na tecnologia de container, uma das pionerias a popularizar a tecnologia, principalmente nesta abordagem onde se virtualiza apenas o ambiente restrito de uma ou outra aplicação específica. Esta tecnologia auxilia na racionalização do uso de equipamentos de infraestrutura de tecnologia por permitir que vários serviços sejam inicializados quando necessário e baixados quando não são mais interessantes, desta maneira auxiliando na redução do desperdício e da ociosidade de *hardware*.

Segundo The Linux Foundation (2016), em junho de 2015 foi criado o projeto OCI (*Open Container Initiative*) pela Docker e outros líderes do mercado de containers, sob os cuidados da Fundação Linux (THE LINUX FOUNDATION, 2017),

que é uma estrutura aberta de governança com o objetivo de criar padrões de formatos e de *runtime* de containers de código aberto.

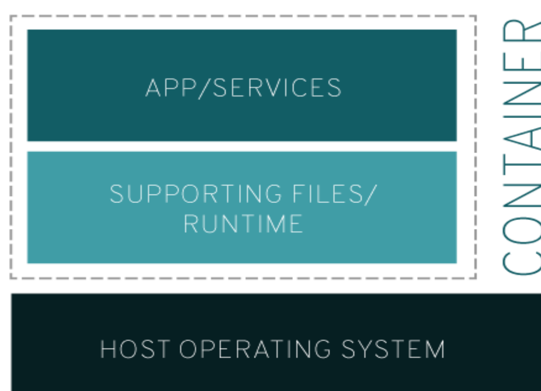
A tecnologia de containers no sistema operacional Linux, especificamente, está no cerne deste trabalho e por isso será abordada em detalhes no capítulo 4.

4 CONTAINERS EM LINUX

Segundo Redhat ([s.d.]) e Kerrisk (2013), a ideia de se criar ambientes virtuais seguramente isolados para a execução de processos dentro do mesmo sistema, foi trazida pioneiramente por Jacques Gélinas para o sistema operacional Linux, através do projeto VServer, no ano de 2001. Gélinas teria dito que o objetivo fora prover uma maneira de executar "vários servidores Linux de uso geral em uma única caixa com um grau elevado de independência e segurança". A ideia evoluiu ao longo do tempo e outras tecnologias foram acopladas, incluindo controle de grupos de processos (**cgroup**), o sistema de inicialização (**Systemd**) que auxilia o controle de grupos e o sistema de mapeamento de espaços de nomes (**namespaces**), até tal evolução chegar ao que hoje é a tecnologia container Linux.

Ainda segundo Redhat ([s.d.]), um container **Linux** é um conjunto de processos isolados do restante do sistema, executados a partir de uma imagem distinta que fornece todos os arquivos necessários para esses processos. Redhat ([s.d.]) também corrobora que por fornecer uma imagem que contém todas as bibliotecas e partes de *software* de que depende um aplicativo, e mesmo arquivos de suporte e configuração de ambiente, o container é portátil e consistente, ou seja pode ser levado para qualquer outro local hospedeiro e utilizado normalmente. A figura 9 ilustra o empilhamento das camadas utilizadas no container Linux.

Figura 9 – Container Linux



Fonte: Redhat ([s.d.])

Por fim, Redhat ([s.d.]) explica que o projeto Linux Containers (LXC) (CANONICAL, [s.d.]) contribuiu com a evolução da tecnologia, melhorando a experiência do usuário, uma vez que agrega ferramentas, bibliotecas, modelos e

API's. Nesse sentido, Redhat ([s.d.]) também enfatiza a capacidade de containers Linux para a montagem de microsserviços:

> O foco dos containers Linux está na capacidade de desenvolver aplicativos com mais rapidez e atender às necessidades da empresa à medida que elas surgem, e não o software usado para realizar esse trabalho. Talvez você acredite que cada container deva conter um único aplicativo inteiro. No entanto, é possível usar os containers para isolar partes de aplicativos ou serviços. Portanto, você pode usar outras tecnologias, como o Kubernetes, para automatizar e orquestrar os aplicativos em containers. Um container hospeda a lógica, o ambiente de execução e as dependências do aplicativo. Por isso, o container pode incluir tudo ou você pode criar um aplicativo composto de vários containers que funcionam como microsserviços. (REDHAT, [s.d.])

A tecnologia **Kubernetes** citada por Redhat ([s.d.]) refere-se a uma plataforma de gerenciamento de containers que permite provisionamento, construção, funcionamento, desligamento e destruição de containers de aplicativos ou microsserviços conforme a demanda, permitindo alto grau de escalabilidade, voltada para ambientes onde existem necessidades maiores de flexibilidade para atender uma demanda muito flutuante. A tecnologia é útil quando há a necessidade de instanciação de muitos containers simultaneamente, em vários locais remotos e em quantidades que inviabilizam o gerenciamento manual (KUBERNETES, 2017).

Grattafiori (2016, p. 13) no entanto, prefere enfatizar que os containers, de uma maneira geral e ainda que de uma maneira rudimentar, surgiram em 1979 na forma de um simples comando "**chroot**" implementado naquele ano em uma versão do sistema operacional Unix (THE OPEN GROUP, 2017) versão 7 e posteriormente em outro sistema derivado do Unix, o BSD (BSD.ORG, 2011). O sistema tinha limitações e a segurança era relativamente fraca e por isso fora quebrada rapidamente. Posteriormente Grattafiori (2016, p. 13) cita outros sistemas que representaram a evolução da tecnologia de containers na linha do tempo como o FreeBSD Jails, Linux VServer, Linux OpenVZ e UML (User Mode Linux). Entretanto, VServer que surgiu em 2001, foi a tecnologia de ponto de partida para a maioria do que existe hoje de moderno sobre containers no *kernel* Linux. Embora tivesse apresentado avanços, a segurança de VServer ainda não era satisfatória, vez que a implementação de espaços de usuários ainda era precária e o controle de grupos de processos inexistia naquela época.

Grattafiori (2016, p. 18) explica que a tecnologia moderna de containers Linux é composta por 5 (cinco) principais componentes: namespaces (espaços de

usuários), cgroups (grupos de controle), root Capabilities (capacidades do usuário root), pivot_root (chamada de sistema para pivot_root) e MAC (Mandatory Access Controls ou controles de acesso obrigatórios). A Canonical ([s.d.]b) e UOL (2017) acrescentam à lista ainda os recursos chroot e Seccomp *policies* do *kernel*, este último como um recurso de segurança não obrigatório.

4.1 chroot

Segundo Free Software Foundation (2017), chroot é um comando utilizado para executar um outro comando passado como parâmetro, porém utilizando um diretório raiz diferente do sistema, também especificado pelo usuário, limitando o acesso do usuário apenas à nova hierarquia de diretórios. Quando se pede para executar um comando com vínculo dinâmico (um programa que depende de bibliotecas externas para ser executado), é necessário que se copie previamente os arquivos de que tal comando dependa para uma estrutura de diretórios idêntica, criada dentro do diretório que será indicado como nova raiz do chroot.

Edge (2007) explica que chroot modifica a maneira como os processos fazem a chamada para a raiz (*root*), acrescentando antes de cada chamada o caminho fornecido como *newroot*. As funcionalidades de chroot são chamadas prisão (*jail*) em sua “man page” (LINUX MAN-PAGES PROJECT, 2017, s. 2). Essa denominação metafórica parece apropriada, visto que as funcionalidades de chroot são semelhantes a uma prisão para os usuários e processos na questão da visão e acesso a hierarquia de diretórios dentro do sistema. Entretanto esta denominação contribui para que pessoas pensem, de maneira equivocada, que a chamada “*jail”* de chroot seria a mesma *jail* utilizada nos sistemas BSD's e derivados (como o FreeBSD Jails). Outrossim, as prisões do chroot guardam pouca semelhança com as prisões do sistema Jails dos sistemas operacionais BSD's. Isso porquê, o segundo é mais sofisticado, tem mais funcionalidades agregadas e trabalha como uma virtualização, o que não ocorre no caso das funcionalidades de chroot.

4.2 Namespaces

Grattafiori (2016, p. 18, 20 – 26) explica que o principal componente de containers em Linux é uma função do *kernel* que promove um isolamento fundamental conhecida como **namespaces**. O conceito é comum em ciência da computação e a ideia foi concebida ainda em 1992.

Explica que durante a criação dos processos (chamada de sistema “clone”), a estrutura de namespaces utiliza-se identificadores *flag* com o prefixo CLONE_NEW. Ele cria separadamente uma nova estrutura de identificação para os processos, usuários, pilhas de rede e outros componentes em cada espaço de usuário, dando a ele uma visão única.

Kerrisk (2013) afirma que um dos objetivos de namespaces é ajudar na construção do sistema de containers e para isso implementa seis tipos diferentes de espaços de nomes. Entretanto, é possível verificar na página manual da chamada de sistema operacional “clone” (LINUX MAN-PAGES PROJECT, 2017, s. 2, p. clone) que, a partir da versão 4.6 do *kernel*, existem 7 (sete) diferentes tipos de namespaces. Kerrisk (2013) explica que cada tipo de espaço de nome envolve um recurso de sistema para que ele se perceba como único em cada instância de namespace. Segundo as definições apresentadas por Kerrisk (2013), complementadas pela página manual de namespaces (LINUX MAN-PAGES PROJECT, 2017, s. 7, p. namespaces), os tipos de namespaces atualmente implementados são os seguintes:

- Mount namespaces – ou namespaces de montagem – utiliza o identificador CLONE_NEWNS (que significa genericamente “novo namespace”) e foi a implementação do primeiro tipo de namespaces que surgiu, ainda no ano de 2002 na versão 2.4.19 do *kernel* Linux. Kerrisk (2013) especula que aparentemente as pessoas que nomearam essa funcionalidade, na época, talvez não imaginariam o surgimento posterior de outros tipos de namespaces. Explica ainda que namespaces de montagem modificou as chamadas mount() e umount() para que os processos pertencentes a cada espaço de nomes tenham uma visão privada da hierarquia do sistema de arquivos. A funcionalidade é semelhante a das prisões feitas pelo chroot, porém mais segura e flexível. Assim, os processos de um determinado namespace tem seus próprios pontos de montagem em uma visão própria do ambiente não conseguindo enxergar, entretanto, as montagens de outras instâncias de namespaces, tampouco podem ter uma visão global do sistema, embora seja possível replicar uma mesma montagem em vários namespaces quando o sistema for externamente instruído para isso.

- UTS namespaces – implementado na versão 2.6.19 do *kernel*, utiliza o identificador CLONE_NEWUTS. Kerrisk (2013) explica que este namespace tem por função tornar privado para um espaço de nomes os identificadores de sistema que referem-se ao nome do *host* e ao nome do domínio do *host*. O objetivo é permitir que seja definido para os processos de cada namespace uma visão particular de nome e domínio do sistema. Proporciona aos processos de determinado namespace a ilusão de que estão em um *host* ou domínio específico. O nome UTS deriva de uma estrutura definida como *Unix Time Sharing System* que é passada quando se faz a chamada uname (LINUX MAN-PAGES PROJECT, 2017, s. 2, p. uname) do sistema.
- IPC (*Inter-process communication*) namespaces (namespaces para a comunicação entre processos) – implementado na versão 2.16.19 do *kernel*, utiliza o identificador CLONE_NEWIPC. O objetivo é promover o isolamento de recursos de comunicação entre processos do System V (SOBELL, 1995) e, após a versão 2.16.30, das filas de mensagens POSIX (SOBELL, 1995), criando um conjunto particular de identificadores para estes objetos em cada namespace. Grattafiori (2016, p. 21) ressalta que com essa funcionalidade tais objetos de comunicação estariam visíveis apenas para processos membros do mesmo userspace, situação necessária para isolamento nos casos de processos que utilizam memória compartilhada. Tal funcionalidade ajuda a prevenir casos de ataques conhecidos como ataques de negação de serviço (DDoS). Kerrisk (2013) explica que para tal isolamento é necessário a implementação deste namespace em particular, porquê para a identificação desses processos de comunicação, são utilizadas técnicas diferentes daquelas que utilizam os caminhos de sistema de arquivos (base dos sistemas Linux, Unix e derivados).
- PID namespaces – implementado na versão 2.6.24 do *kernel*, utiliza o identificador CLONE_NEWPID. O objetivo deste namespace é isolar a sequência de numeração para identificação dos processos. Kerrisk (2013) explica que para cada namespace poderá haver um processo diferente com PID 1, por exemplo. Ele ainda alerta que, embora a numeração dos processos seja reinicializada em cada namespace, é importante ter em vista que em container todo o processo, na prática, é executado no tempo

compartilhado do *kernel* principal (*host*) e por isso ele também terá um número PID próprio no sistema principal, ou seja, seguindo a sequência original do sistema hospedeiro. Assim, há uma ligação de equivalência entre o PID criado no sistema hospedeiro e o PID criado e visualizado dentro do container, pois na prática ambos se referem ao mesmo processo. Isso ajuda no caso de um container que poderá ser migrado de um *host* para outro sem que haja interferência no funcionamento dos sistemas. Os PID's internos de cada processo no interior do container permanecem o mesmo, mas o novo sistema hospedeiro pode assumir novos números de PID fora do container para cada processo migrado, conforme lhe for conveniente. A ideia pode ser explicada por uma suposição onde um computador físico A que já tenha inicializado outros processos e onde o último processo inicializado tenha o PID número 499, se inicialize também um container C e que o primeiro processo criado dentro do container seja um processo “init” e que por ser o primeiro ele receba no interior do container como PID o número 1, temos que dentro do sistema hospedeiro, ao contrário, este mesmo processo receberá o PID 500. Então, supondo também que este container seja migrado para um outro computador físico, um sistema hospedeiro B que já teria previamente 650 processos inicializados antes da migração, o processo init dentro do container continuaria com o PID interno sendo o de número 1, porém no novo sistema hospedeiro esse processo passaria a ter o número de PID com sendo 651 e não mais 500, pois ele seguiria o número de sequência disponível para ele no novo hospedeiro.

- Network namespaces – com a implementação inciada a partir da versão 2.4.19 (concluído na versão 2.6.29) do kernel, utiliza o identificador CLONE_NEWNET. O objetivo é isolar os recursos de rede para cada userspace, possibilitando que possa ter virtualizado seus dispositivos de rede, endereçamento IP, tabelas de roteamento IP, o próprio diretório /proc/net, portas, enfim toda a estrutura de rede. Assim, poderia existir dentro de um mesmo computador hospedeiro vários containers, cada um contendo um endereço IP diferente com um servidor Web ouvindo sua própria porta 80 sem qua haja conflito entre eles. E todos podem se comunicar através da mesma interface física de rede.

- User namespaces (namespaces de usuários) – com a implementação iniciada na versão 2.6.23 e finalizada na versão 3.8 do *kernel*, utiliza o indentificador CLONE_NEWUSER. O objetivo deste namespace é criar um isolamento para os números de usuários e para os números de grupos de usuários, permitindo a cada namespace que possa iniciar seus próprios usuários e grupos com sua numeração sequencial própria. Segundo Kerrisk (2013), o resutado disso é que um processo iniciado em um namespace estará vinculado a um número de usuário e a um número de grupo dentro do ambiente do namespace, porém poderá estar vinculado a outro número de usuário e ainda a outro número de grupo fora do ambiente do namespace (no sistema hospedeiro). Isso significa que um processo que estiver vinculado a um usuário com privilégios limitados no sistema hospedeiro, tendo assim acessos limitados neste ambiente, porém iniciado dentro de um namespace e nele estando vinculado a um *id* (número identificador) de usuário *root* (com privilégio máximo) terá este processo, dentro do ambiente do namespace, o privilégio referente a este usuário *root*, ou seja, privilégio máximo. Em outras palavras, o processo terá todos os privilégios e permissões dentro do ambiente do namespace, porém privilégios limitados no sistema hospedeiro ou fora do namespace. Embora o esforço para tornar real e efetiva esta funcionalidade tenham sido grande, as mudanças são grandes e complexas de modo que pode ser que falhas de segurança ainda sejam encontradas.
- CGroup namespaces (namespaces de grupos de usuários) – Implementado na versão 4.6 do *kernel*, utiliza o identificador CLONE_NEWCGROUP (LINUX MAN-PAGES PROJECT, 2017, s. 7, p. cgroup_namespaces). Segundo Kerrisk (2013), o objetivo deste namespace é isolar a hierarquia de diretórios especiais de cgroups (LINUX MAN-PAGES PROJECT, 2017, s. 7, p. cgroups). Assim, cada instância de namespace passa a ter um diretório raiz do cgroup próprio, sendo a partir dele, configurado de maneira isolada, tendo então uma visão particular das configurações cgroup de cada processo. Quando um namespace de cgroup é criado, a raiz de cgroup para aquela visão de namespace é montada a partir do diretório cgroup do processo chamador.

Kerrisk (2013) explica que hoje em dia, após muitos anos desde a primeira implementação de namespace, o conceito se expandiu, tornando privado de cada namespace o escopo de muitas funcionalidades que antes eram de um escopo geral do sistema, permitindo assim a base necessária para criação de uma virtualização leve por meio de containers.

4.3 Os cgroups

A ligação existente entre containers e cgroups é de concepção tão íntima que Grattafiori (2016, p. 18, 27 – 29) conta que o próprio cgroups se originara como um método apresentado como “Process containers”, pelos engenheiros do Google em 2006, ainda quando o *kernel* estava na versão 2.6.24.

Explica Grattafiori (2016, p. 18, 27 – 29) que no contexto de containers, cgroups é uma funcionalidade utilizada para controlar o acesso a recursos, podendo prevenir processos defeituosos ou maliciosos que possam tentar instruir o sistema de forma errônea a promover o consumo inadequado de recursos de forma genérica em situações onde muitas vezes ocorre a degradação do desempenho do sistema em parte significativa dele. Problemas em geral como encadeamentos programados de maneira desenfreada que excedem o uso de memória RAM, *forkbombs* (STATO, 2016) e ataques de negação de serviço (SYMANTEC, 2017) são exemplos deste tipo de problema que pode ter seu efeito, ou a possibilidade de ocorrência, reduzida pelo uso de cgroups. Ele explica que cgroups também pode ser utilizado para limitar o acesso a alguns dispositivos com auxílio de uma uma lista contendo informações sobre outros dispositivos, ou seja, apenas os que estariam com acesso liberado. Seria no caso, uma espécie de “lista branca”, onde pode-se indicar os dispositivos liberados para uso.

Na visão de Grattafiori (2016, p. 18, 27 – 29), cgroups é um mecanismo que age com alguns subsistemas relacionados para limitar um processo ou um conjunto de processos quanto à utilização ou mesmo quanto ao acesso ao *hardware*, apresentando como exemplo o tempo de processamento da CPU, quantidade de uso da memória RAM e desempenho de disco rígido, visando controle de desempenho ou segurança. Para isso ele cria grupos de controle que funcionam através de um sistema de arquivos especialmente montado para isso. Outras ferramentas podem utilizar este sistema de arquivos com intuito de visualizar ou mesmo de controlar namespaces e control groups. Se o sistema não estiver bem

configurado, com as devidas restrições ao acesso deste sistema de arquivos, estando ele montado internamente em um container poderá servir como um escape do container, comprometendo então a segurança do sistema.

Grattafiori (2016, p. 18, 27 – 29) explica ainda que a funcionalidade evoluiu ao longo do tempo, sendo que com tal evolução passou gradativamente a oferecer maneiras para se extrair dados estatísticos, controle de acesso obrigatório (MAC), limites de criação para novos PID's, controles de entrada e saída de dados, controles de rede como níveis de prioridade de tráfego aplicados pelo Iptables (DIE.NET, [s.d.], s. 8) e limites de buffer, congelamento de tarefas, tudo isso podendo ser realizado de maneira específica por grupo controlado.

Grattafiori (2016, p. 18 – 29) explica que, assim como namespaces, cgroups também trabalha de maneira hierárquica e herdável, podendo ainda ser aninhado em um modelo de árvore.

4.4 Capabilities para o usuário root

Segundo a Canonical ([s.d.]b), quando o usuário *root*, aquele de identificador 0 (zero) de um dado container é mapeado para o usuário de identificador 0 (zero) do sistema hospedeiro (o *root* do *host*), temos que o container é chamado de **container privilegiado**. São nessas situações que torna-se necessário para a segurança do sistema tomar providências para limitar e controlar os recursos do usuário *root*, sem que o container deixe de ser privilegiado, já que existem situações onde eles são necessários.

Grattafiori (2016, p. 18, 30 – 42) explica que *capabilities* é um recurso utilizado para limitar as capacidades de usuários, sendo que quando aplicado ao usuário *root* (raiz) no sistema, torna-se útil para reforçar os objetivos dos namespaces em sistemas de containers quando impõe limite às capacidades deste usuário especial. Isso é mais um mecanismo de segurança que serve também para limitar o poder dos containers privilegiados, dentro do ambiente hospedeiro. O usuário *root* normalmente é detentor de todos os privilégios para fazer qualquer operação com o sistema *host* e com esta funcionalidade ele pode ter suas capacidades reduzidas a zero, por exemplo, dentro de um ambiente de container.

4.5 pivot_root

Para Grattafiori (2016, p. 18, 30 – 42), pivot_root é uma chamada de sistema (*syscall*) utilizada para alterar o sistema de arquivos raiz e pode ser crucial para a segurança em ambiente de container.

Linux man-pages project (2017, s. 2), explica que pivot_root é uma *syscall* (chamada de sistema operacional) enquanto Linux man-pages project (2017, s. 8) apresenta-o como um comando para executar a mesma funcionalidade. O objetivo desta é guardar as informações sobre o sistema de arquivos do diretório *root* (raiz) atual do processo que invocou a aludida funcionalidade, e posteriormente modificar o diretório *root* (raiz) do processo em execução para apontar um novo sistema de arquivos passado como parâmetro. Assim, pivot_root pode não alterar o diretório de trabalho atual de um processo em funcionamento que utilize o diretório antigo. Quem deve fazer isso, como segurança, é o usuário que chama pivot_root, podendo para isso, por exemplo, modificar o diretório de trabalho atual para o novo diretório antes mesmo de se invocar a funcionalidade pivot_root.

Segundo se confere no código fonte de pivot_root em Kernel (2017) e de chroot em Kernel (2017b), estes dois comandos tem semelhanças, embora pode-se observar que pivot_root faz uma espécie de nova “montagem” da raiz do sistema, sendo que neste caso não permanece nenhuma ligação que possa permitir ao processo escalar e chegar na raiz antiga do sistema. No caso de chroot, ao contrário, este modifica a raiz para o processo, porém, embora as chamadas do processo invoquem sempre a nova raiz que chroot remete à antiga raiz para encontrar o caminho do *newroot*, mantendo assim um vínculo que pode ser explorado pelo processo para acessar diretórios e arquivos que estão em outra raiz.

4.6 Mandatory Access Control (MAC).

Implementados pelas funcionalidades AppArmor (DEBIAN, [s.d.], s. 14.4) e SELinux (DEBIAN, [s.d.], s. 14.5), segundo Grattafiori (2016, p. 18, 69 – 42) os controles de acesso obrigatórios (MAC), embora não sejam obrigatórios para a implementação de containers, são importantes para reforçarem a segurança de containers de uma maneira mais profunda, promovendo também a segurança da plataforma como um todo, containers, usuário privilegiado, etc. A Canonical ([s.d.]b) também vê este recurso como um instrumento de poda de poderes para os processos, interessante para a segurança de containers privilegiados.

Hess (2010) elenca como sistemas MAC utilizados no Linux atualmente, além de AppArmor e SELinux, também os sistemas SMACK (LINUX KERNEL ORGANIZATION, 2017b) e TOMOYO (LINUX KERNEL ORGANIZATION, 2017c). Ele relata que os sistemas MAC podem inclusive ajudar a tornar os sistemas Linux mais seguros contra falhas do próprio sistema.

Embora os controles de acesso obrigatórios (MAC) sejam importantes aditivos de segurança para os ambientes de container, eles não estão no escopo deste trabalho e portanto não serão abordados em detalhes.

4.7 Seccomp *policies*

Segundo a Canonical ([s.d.]b) Seccomp é um recurso não obrigatório para a implementação de containers, porém importante para a segurança deles, principalmente quando se fala em containers privilegiados.

Seccomp *policies* é um recurso de segurança que reduz a área de exposição da superfície do *kernel* (LINUX KERNEL ORGANIZATION, 2017d). Ele promove uma maneira para que o próprio processo especifique uma filtragem de chamadas de sistemas para serem realizadas ao *kernel*. Assim, observando de outro ângulo, aquele determinado processo pode delimitar para si as chamadas de sistema do *kernel* a que ele estará exposto, além de dizer também o número de parâmetros passados para cada chamada, evitando assim que algum erro ou alguma exploração indevida faça uma chamada desnecessária e até indesejada ao *kernel*. Embora este recurso seja mencionado por autores como um importante recurso de segurança para containers, ele não será abordado em detalhes pois está fora do escopo do trabalho.

4.8 Eliminando a influência automática de root

Grattafiori (2016, p. 42) explica que o Linux adicionou um conjunto de bits de segurança para processos que podem ser utilizados para eliminar privilégios especiais dado normalmente ao usuário identificado como 0 (zero), identificação que equivale ao usuário *root*. São bits alocados para cada *thread* iniciada no sistema. Embora existam vários níveis de capabilities para qualquer usuário, para remover completamente os privilégios automáticos de *root* é necessário apenas o uso do bit de flag SECBIT_NOROOT. Desta maneira aumenta-se o controle do ambiente exigindo-se que as permissões de *root* também tenham que ser concedidas, eliminando a presunção automática de que esse usuário poderia fazer qualquer

coisa. É muito importante tomar-se o cuidado de não dar direitos em executáveis privilegiados para usuários não privilegiados, pois assim se evita que este tenha a possibilidade de encontrar uma brecha.

4.9 LXC – Projeto Linux Containers

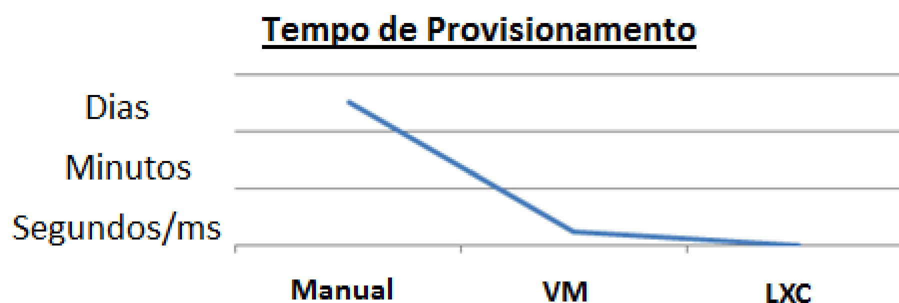
Segundo Redhat ([s.d.]) e Canonical ([s.d.]), LXC (ou projeto Linux Containers) é uma interface construída com uma API e um conjunto de ferramentas, utilizada no espaço de usuário para o controle e o gerenciamento dos recursos relacionados a containers (tanto para sistemas quanto para aplicações) no *kernel* Linux. Os autores corroboram a ideia de que tais ferramentas facilitam a vida do usuário neste tipo de trabalho.

Canonical ([s.d.]) explica que o projeto LXC é atualmente composto por alguns componentes:

- A biblioteca liblxc;
- API's para algumas linguagens de programação;
- Um conjunto de ferramentas padrão para controlar os containers;
- *Templates* (modelos) de container.

Segundo Uol (2017), LXC é uma plataforma que fornece uma das melhores maneiras de se conseguir segurança, desempenho e agilidade na criação de aplicações. O provisionamento é instantâneo e dinâmico, além de oferecer desempenho para as aplicações muito semelhante ao do servidor físico. Sua aceitação vem crescendo muito no mercado, sendo que este tipo de tecnologia tem sido a base dos ambientes de *cloud computing* atuais. Neste tipo de ambiente altamente dinâmico, há necessidades de provisionamento rápido é constante:

Figura 10 – Provisionamento de recursos no LXC.



Fonte: Uol (2017)

Uol (2017) ilustra na figura 10 que o tempo de provisionamento de recursos no LXC é extremamente pequeno, mesmo quando comparado às máquinas virtuais.

4.10 Plataforma Docker

A plataforma de containers Docker pode oferecer suporte aos containers contendo todo um sistema operacional completo como também aos containers contendo apenas um aplicativo ou até mesmo parte de um aplicativo (microsserviço), situações voltadas principalmente para o desenvolvimento e a implantação de softwares (DOCKER, 2017).

Para Docker (2017b), a plataforma oferece sempre o mesmo ambiente para o funcionamento dos aplicativos, independente de onde estejam hospedados, ainda que em um sistema operacional Linux ou mesmo em um sistema operacional Microsoft Windows.

Segundo Docker (2017b), os containers Docker que funcionam hospedados no mesmo hospedeiro compartilham o mesmo *kernel*. Assim os containers iniciam de forma instantânea, minimizando o acesso a disco e uso de processamento e memória RAM. O sistema também economiza recurso de espaço em disco compartilhando arquivos comuns entre os containers.

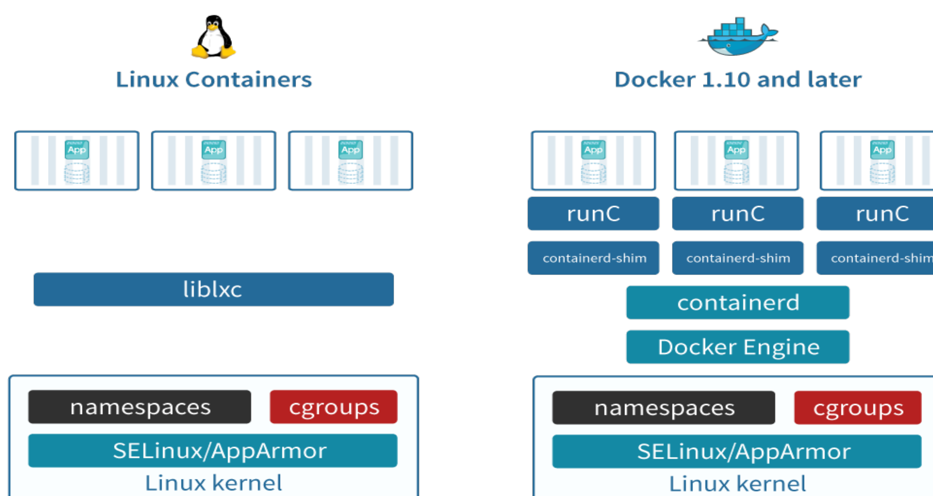
Redhat ([s.d.]) explica que Docker foi construído como uma mescla da tecnologia de LXC com outras ferramentas:

> Em 2008, o Docker entrou em cena (por meio do dotCloud) com sua tecnologia de container homônima. A tecnologia docker é uma combinação do trabalho do LXC com as ferramentas aprimoradas para desenvolvedores, aumentando, assim, a facilidade da utilização dos containers. O docker, uma tecnologia open source, é atualmente o projeto e o método mais famoso para implantar e gerenciar containers Linux. (REDHAT, [s.d.])

Fulay (2017) explica que as primeiras versões de Docker foram implementadas literalmente sobre a plataforma do LXC que era seu ambiente de execução padrão. No entanto, desde o início o foco da plataforma sempre foi o de prover suporte voltado para desenvolvedores de software que normalmente utilizam *laptop* além de suporte a todas as distribuições Linux. No intuito de cumprir esta meta, Docker se distanciou de LXC e em abril de 2014 lançou sua própria biblioteca para gerenciamento dos recursos de container, a libcontainer, abandonando a biblioteca liblxc que até então era utilizada como padrão no projeto Docker. No decorrer dos meses seguintes foram implementadas múltiplas camadas de abstração, algumas mudanças no projeto para o cumprimento dos padrões da OCI

sendo que em 2015 colocaram runC e containerd como sendo os principais componentes do *engine* (motor) Docker. As semelhanças e diferenças que restaram entre as plataformas podem ser vistas na figura 11:

Figura 11 – LXC vs. Docker



Fonte: Fulay (2017)

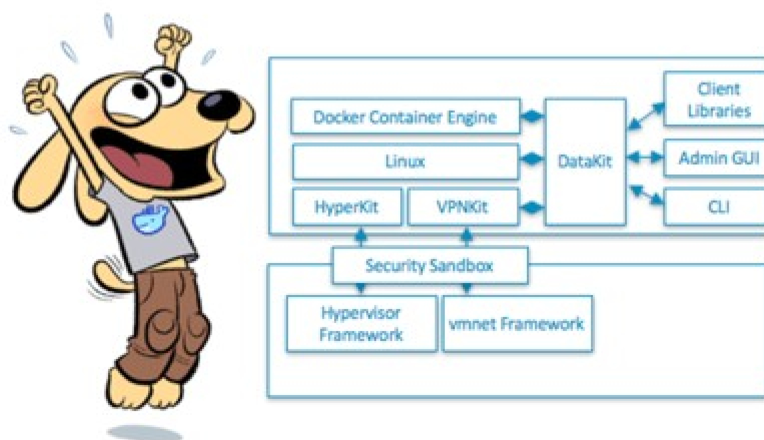
Na figura 11, ambas as plataformas apresentam a base de containers em Linux, porém a camada de abstração de Docker contém atualmente seus componentes próprios em número maior.

Verifica-se com as informações de Docker (2017b) e Fulay (2017) que ainda antes da implementação de libcontainer, Docker já era uma plataforma que vinha experimentando um crescimento muito forte na quantidade de usuários. Com as mudanças promovidas na plataforma, esta foi capaz de sustentar um crescimento vertiginoso levando-o a estabelecer uma grande distância na liderança de mercado sobre o provimento de containers em relação à concorrência. Finalmente, Docker passou a ter suporte também na plataforma Windows.

Segundo Docker (2017c, 2017d), para a plataforma funcionar no Microsoft Windows porém, Docker para Windows (como é chamada) se apresenta como uma ferramenta nativa daquele sistema operacional, que para executar mecanismos tradicionais de container, se integra profundamente com a ferramenta de virtualização *hypervisor* daquele sistema operacional, o Microsoft Hyper-V. E da mesma maneira ocorre com a plataforma no sistema operacional MacOS, onde a plataforma denominada Docker para Mac precisa se integrar com a estrutura do *hypervisor* MacOS que é a ferramenta de virtualização neste caso, dando suporte

também para o modelo de segurança Sandbox daquele sistema operacional. Docker (2017c) apresenta a figura 12, que abaixo ilustra essa estrutura:

Figura 12 – Docker para Windows



Fonte: Docker (2017c)

Desta forma, Docker se apresenta como uma solução de suporte aos containers que os tornam portáveis, podendo levar a mesma aplicação montada inteira dentro de um container, ou montada sobre uma interconexão de vários containers (cada container rodando um microsserviço), a funcionar hospedado em qualquer ambiente hospedeiro, seja ele com sistema operacional Linux, Windows ou MacOS, ou uma mescla deles, sem a necessidade de novas adaptações pelo desenvolvedor, situação muito positiva para ambientes mistos, tanto quando falamos do caso de times de desenvolvimento de sistemas quanto para ambientes de *cloud computing*.

5 CENÁRIO: PROJETO DE ALTA DISPONIBILIDADE MONTADO EM CONTAINERS

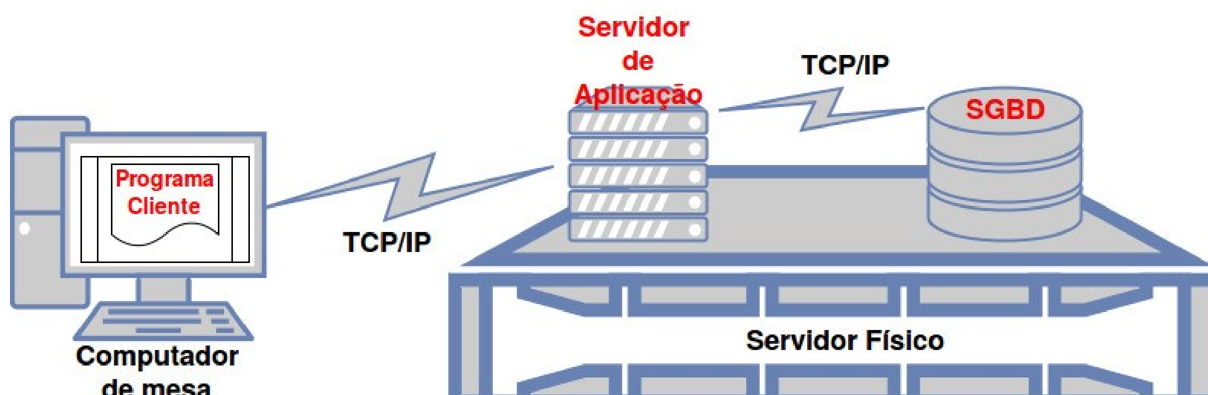
O estudo de caso é de uma empresa de pequeno porte (classificada fiscalmente como EPP), que trabalha com comércio varejista de tintas, tendo três lojas (uma matriz e duas filiais) distribuídas pelas cidades da região. Trata-se de uma empresa real, embora informações sobre sua identificação e detalhes do caso concreto estejam omitidos do trabalho por uma opção da mesma.

Esta empresa possui um sistema de informação que auxilia os usuários no gerenciamento do cadastro de clientes, fornecedores, funcionários em geral, vendedores especificamente, produtos para revenda, controle de estoque dos produtos, controle de vendas, controle de cobranças e controle financeiro.

Todas estas informações são controladas por este sistema de maneira integrada, sendo que as informações de um departamento interagem com as informações de outro. Este sistema, para efeitos deste trabalho, é chamado simplesmente de “ERP”. Segundo as políticas de segurança da informação da empresa, os usuários precisam ter acesso cotidianamente ao ERP, mormente no horário comercial que compreende de segunda às sexta-feira das 8h00 às 18h00 e aos sábados das 8h00 às 12h00, sendo que a indisponibilidade deste sistema, ou o comprometimento da disponibilidade ou da integridade das informações nele contidas, pode causar prejuízos para a empresa.

O ambiente em questão é composto por um sistema distribuído, onde terminais de acesso (computadores de mesa) executam um programa “cliente”, multiplataforma, escrito em linguagem Java, que se conecta a um programa “servidor de aplicação” utilizando um protocolo de comunicação proprietário via rede (*socket* Ip). Este programa cliente realiza a tarefa conhecida como *front-end*, ou seja, exibe (tela/monitor) ou imprime (papel/impressora) as informações do sistema diretamente conforme a consulta solicitada pelo usuário, além de servir de entrada para os dados digitados para serem enviados ao servidor de aplicação, que por sua vez fará o gerenciamento das regras de negócio e por fim, fará a comunicação com um sistema gerenciador de banco de dados (SGBD) PostgreSQL para a gravação centralizada e definitiva dos dados. Tanto o servidor de aplicação quanto o SGBD estão instalados e funcionando em um único servidor físico central. A figura 13 ilustra esse cenário:

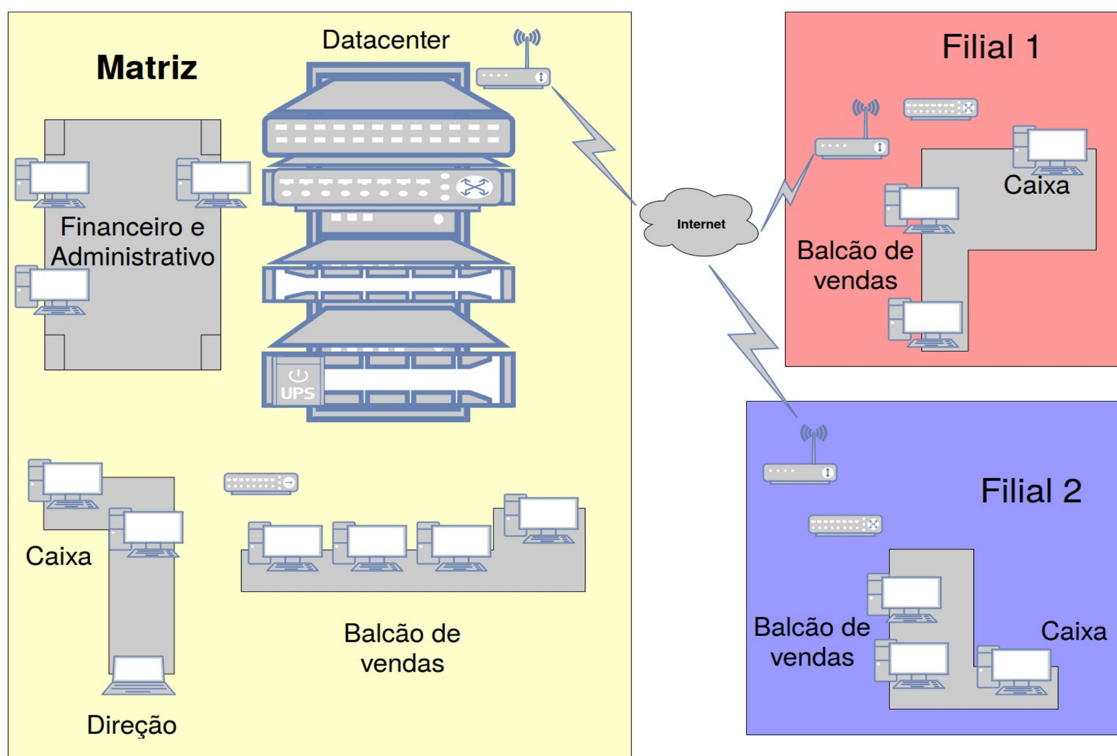
Figura 13 – Sistema do cenário



Fonte: próprio autor

Para o funcionamento de tal sistema, existe atualmente um conjunto composto de um total de dezesseis microcomputadores *desktop* (de mesa) ou *laptop* (portátil) que são utilizados como terminais de consulta, tendo em vista que, em sete deles o sistema operacional utilizado é o Microsoft Windows 7 64 Bits, e que em outros nove é utilizado o sistema operacional Linux Ubuntu 64 Bits para *desktop*. A figura 14 ilustra este cenário:

Figura 14 – Cenário físico



Fonte: próprio autor

Nas instalações físicas da matriz, existem 9 (nove) computadores de mesa, sendo que os 4 (quatro) computadores do balção de vendas e os 2 (dois) computadores do caixa, além do *laptop* utilizado pela direção, estão ligados ao *datacenter* através de um *switch* de 8 (oito) portas. No setor administrativo e financeiro existem outros 3 (três) computadores de mesa que são ligados diretamente ao *datacenter*. No *datacenter* existe um *switch* central de 8 (oito) portas, e a infraestrutura de comunicação com a internet e rede sem fio, além do servidor e um sistema de fornecimento de energia elétrica initerrupta (*UPS*).

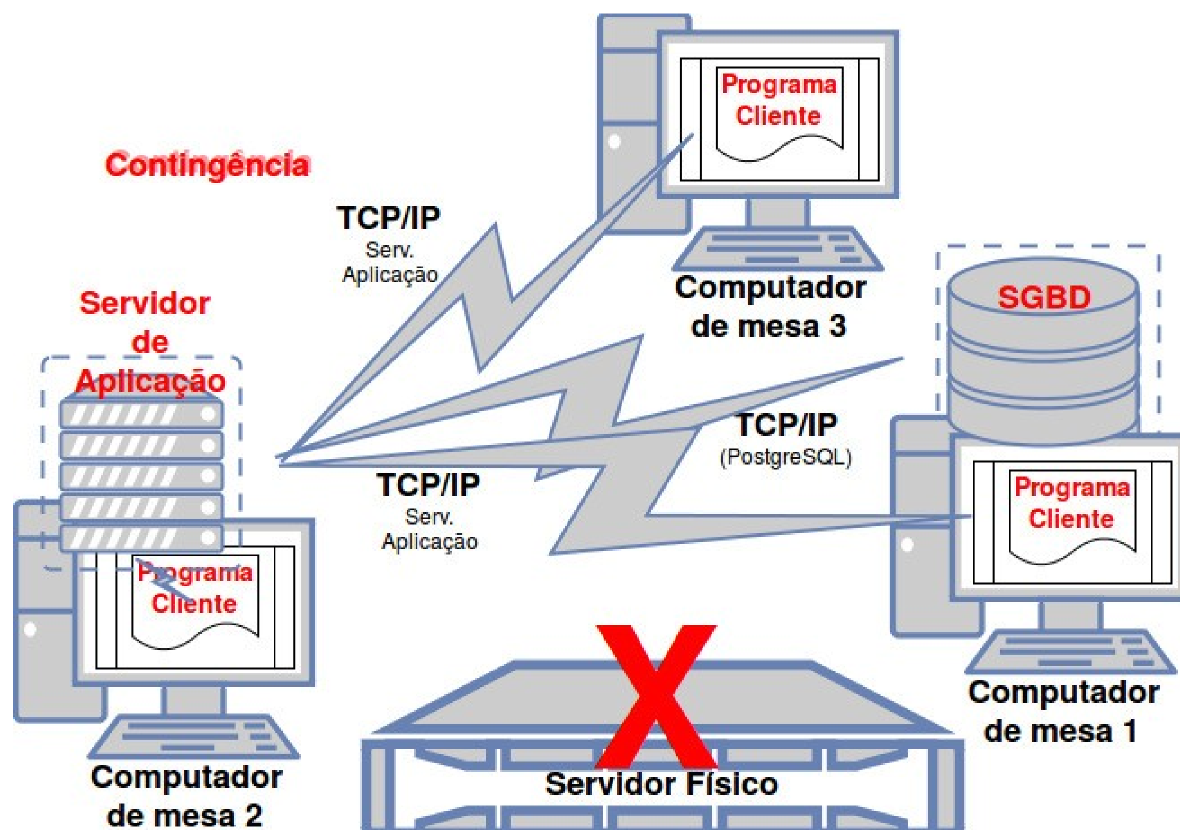
As lojas também utilizam *switch* de 8 portas para distribuição da rede. A empresa utiliza estrutura de rede econômica e semelhante em todas as lojas, além de redundância de *link* de internet na matriz e dispõe de *switch* sobressalente para reposição em caso de algum equipamento apresentar falhas.

O gargalo ocorre no servidor onde se concentra os serviços centrais do ERP, tanto o servidor de aplicação quanto o de banco de dados. Apesar do servidor ser um computador relativamente confiável, a direção reclama que houveram situações em que o *hardware* apresentou falhas e a substituição de peças demorou a acontecer, sendo que durante o período de substituição da peça que apresentara defeito, todas as três lojas da empresa ficaram sem acesso ao sistema de informação, causando prejuízos a empresa. Embora tais prejuízos sejam indesejáveis, a empresa conviveu com o risco de ficar sem o sistema em caso de nova falha no *hardware* do servidor, pois considera o investimento em duplicação do servidor muito elevado.

A solução implementada foi de virtualizar o servidor, retirando os serviços do *bare metal* (servidor físico), passando-os para ambientes virtuais isolados que possam ser iniciados facilmente em outros computadores físicos (no caso os *desktops*) em caso de o servidor físico central apresentar falhas. É sabido que os computadores pessoais de mesa não são tão confiáveis, nem tem tanto desempenho quanto os servidores para executarem serviços grandes e centralizados. Entretanto, a ideia é de executar tais serviços nos *desktops* apenas quando o sistema estiver em regime de contingência após o servidor físico central apresentar uma falha, até que o mesmo seja devidamente reparado ou substituído, para que neste período os serviços permaneçam em funcionamento.

Optou-se pelo sistema de containers por este ser mais leve para funcionar em desktops quando comparado as máquinas virtuais tradicionais, além de permitir que seja levantado nele apenas o *software* em si, sem a necessidade de outro sistema operacional inteiro rodando. O fato de ser uma virtualização leve contribui para que o desktop mantenha suas atividades cotidianas em funcionamento além de manter o container funcionando em segundo plano, sem que o usuário sinta um prejuízo muito significativo no desempenho. Assim, quando houver regime de contingência, enquanto os computadores de mesa executam os containers em segundo plano, seus usuários podem continuar a utilizarem os mesmos normalmente, sendo que um mesmo computador de mesa poderá executar o programa cliente do sistema ERP que se conectará ao serviço do servidor de aplicação onde quer que ele esteja funcionando, desde que a rede esteja acessível. Desta forma, um programa cliente procurará se comunicar com um servidor de aplicação que poderá estar hospedado no mesmo computador físico de mesa (ou ainda em um terceiro), e este servidor de aplicação, por sua vez, procurará se comunicar com o SGBD que estará hospedado em outro computador físico de mesa. A figura 15 ilustra essa situação:

Figura 15 – Sistema de contingência



Fonte: próprio autor.

Na figura 15, observa-se que em uma situação de contigência, um computador de mesa assume o serviço de SGBD, enquanto o outro assume o serviço de servidor de aplicação. Isso ocorre através do acionamento manual de *scripts* nos respectivos computadores de mesa. Situação curiosa é observada no comportamento do computador de mesa 1, quando o programa cliente pede ou envia informações ao servidor de aplicação que está hospedado no computador de mesa 2, e este por sua vez consulta ou envia informações ao SGBD que está no computador de mesa 1. Já no computador de mesa 2, quando o programa cliente faz a consulta ao servidor de aplicação, tal comunicação permanece dentro do mesmo computador físico, pois este hospeda simultaneamente os dois serviços que se comunicam mutuamente. No caso do computador de mesa 3, não há muita novidade, senão pelo fato de que encaminhará suas consultas ao servidor de aplicação, que no regime de contigência encontra-se hospedado em outra máquina física.

Para a configuração deste sistema, optou-se pelo Docker CE como gerenciador de containers por este oferecer maior portabilidade entre as plataformas de sistemas operacionais, quando comparado ao LXC, sendo que no Docker é possível de se rodar o mesmo container tanto em um *desktop* executando o sistema operacional Linux quanto em um outro executando um sistema operacional Microsoft Windows, ou ainda em um MAC-OS, embora a implantação atual tenha sido realizada toda em *desktops* que estão executando sistemas Linux.

Assim, foram criados duas instâncias de containers Docker. Uma instância refere-se a um container Docker montado com a imagem “postgre” para executar o serviço de SGBD, e a outra instância utilizando uma imagem “java:8”, para a execução do servidor de aplicação, ambas baixadas do Docker Hub.

Primeiro foram criados os containers no servidor físico central. O Docker baixa via internet as imagens de containers do Postgresql e do Java automaticamente do Docker Hub.

Também foi criado um “alias” de endereçamento de rede virtual relativo a interface física de rede no sistema operacional, para cada um dos containers, para que estes pudessem ser acessados externamente, independente da máquina que estivessem funcionando, sempre pelo mesmo endereço IP.

A ideia é que estas mesmas configurações de endereçamento de rede sejam realizadas, separadamente, em dois computadores de mesa quando houver algum desastre, para que em cada computador de mesa, em regime de contingência, venha a existir um ambiente com o mesmo endereço IP disponível para o container que nele estiver hospedado.

Para a instalação do sistema PostgreSQL, primeiramente foram instalados dois discos rígidos externos, portáteis de interface USB 3.0. Optou-se por configurá-los em um sistema de RAID nível 1 por *software*, ou seja, um tipo de espelhamento. Isso significa que estes discos foram configurados para serem enxergados como um único dispositivo de bloco de dados, ou seja, um dispositivo único de memória secundária enxergado em um diretório. Os containers foram configurados de tal forma a persistirem seus dados dentro deste diretório, ou seja, dentro dos discos espelhados.

Passou a existir dados do banco de dados PostgreSQL replicados em ambos os discos externos que, por terem interface USB, podem ser fisicamente levados para os computadores de mesa com grande facilidade em casos de desastre.

Foi criado um *script* que, com auxílio da ferramenta cron (um agendador de tarefas) do Linux, executado no servidor *host* central, verifica a cada cinco segundos se os processos referentes ao container de SGBD, bem como ao container de servidor de arquivos, estão em execução no sistema operacional. O *script* tenta subir cada serviço por três vezes consecutivas, fazendo um registro de Log em cada tentativa, mesmo que bem sucedida. Este tipo de estratégia ajuda a manter o serviço disponível em casos de falhas de *software*. Em caso das tentativas de ativação dos processos não surtirem resultados, o *script* encaminha um e-mail para um funcionário treinado e para a equipe de informática avisando do problema com o *software*.

Também foi realizada uma simulação de falha, desligando-se o serviço de SGBD com um comando kill, enquanto um programa cliente realizava uma consulta. Tal simulação fez com que o servidor de aplicação perdesse a conexão com o banco de dados, gerando uma linha de exceção no arquivo de *log* do aplicativo. Entretanto, o *script* recolocou o container de SGBD no ar quase instantaneamente, sendo que o programa cliente não percebeu o problema, senão por um intervalo de quatro

segundos, que foi o tempo necessário para que o resultado da consulta aparecesse na tela.

Há um outro *script* que pode ser acionado, opcionalmente também por intervenção de um usuário, que tenta então inicializar uma instância “reserva” de container idêntica a aquela que não funcionou, todavia esta instância pega os dados de um outro diretório (reserva) contido em outro disco rígido interno do servidor central.

Para viabilizar este procedimento, também foi criado um *script* que realiza a parada dos containers e em seguida realiza uma cópia dos dados para um local separado (reserva), dentro do próprio servidor. Este *script* toma o cuidado de paralisar o aplicativo cron que mantém os serviços ligados antes de prosseguir com a tarefa. Ele é executado uma vez por dia, sendo uma às 0h00, pois neste horário há pouca ou nenhuma utilização do sistema das lojas. A cópia é sincronizada com o aplicativo *rsync* em poucos minutos e em seguida, o *script* coloca o container novamente em funcionamento.

Ao chegarem ao trabalho, os funcionários ligam os computadores de mesa que automaticamente executam outros *scripts* com a finalidade de atualizarem, agora via rede, suas máquinas com os dados do servidor. A atualização remota é feita também com o comando rsync, que trafega as alterações dos arquivos via rede do servidor central para o computador de mesa, o que também ocorre de maneira muito rápida.

Toda a comunicação do rsync que corre através da rede é realizada através do protocolo ssh, sendo que a autenticação da conexão ocorre através de chaves de criptografia trocadas previamente entre os servidores.

Em caso de desastre, optou-se pelo sistema de *switchover*, ou seja, por levantar os serviços manualmente, assim que o problema for detectado por algum usuário da empresa. Foi criado um procedimento para o reestabelecimento do sistema em caso de pane, sendo que o levantamento dos serviços poderá ser realizado com base nos discos externos (recomendado) ou, em último caso, com base nos dados de reserva realizada via rede.

Caso seja percebida a ausência de acesso ao sistema, verificando-se que o servidor encontra-se inacessível, um procedimento de reinicialização do servidor

está descrito como padrão. Em caso de não haver resposta por parte do servidor, um procedimento está descrito para que o usuário, devidamente treinado, pegue um dos discos rígidos externos do servidor e ligue apenas um deles ao computador de mesa. Após, o usuário deve executar um *script* especial que verifica a presença do disco, detecta-os e os configura para executar novamente os containers.

Esse script finaliza o *switchover* permitindo o sistema estar disponível em regime de contingência.

Assim, o SGBD estará funcionando. Caso não consiga obter sucesso com o primeiro disco utilizado, o usuário pode tentar o mesmo procedimento com o outro que é uma cópia idêntica.

Outro computador de mesa dividirá a tarefa executando o servidor de aplicação, sendo que nele um outro *script* será executado, inciando os serviços à partir da cópia reserva do computador. Isso é possível porquê o servidor de aplicação não precisa ter uma cópia muito atualizada, como é o caso do SGBD.

Desta forma, todos os serviços se estabelecem, mesmo que o servidor físico principal venha a falhar e não se reestabeleça. Após a manutenção do servidor principal, um procedimento *switchover* de retorno está previsto para que o sistema retorne ao local original.

Foi realizado um teste comparativo, utilizando-se uma máquina virtual contendo o sistema operacional Ubuntu 16.04 LTS Server básico, hospedado no mesmo computador físico servidor onde encontram-se os containers. Um *script* de teste foi feito para subir a máquina virtual e começar o processo de Ping no endereço IP da máquina virtual. Os pings não obtiveram resposta durante aproximadamente 37 segundos, até que a VM estivesse com o sistema de rede funcionando e pudesse respondê-los.

De outro lado, foi construído um outro *script* para que, em conjunto com uma aplicação Java que faz conexões JDBC ao SGBD no container, pudesse monitorar quanto tempo o SGBD levaria para aceitar a primeira conexão. Desde que o comando para subir o container foi disparado, o tempo até a primeira conexão ser aceita foi inferior a dois segundos, deixando claro a enorme vantagem do sistema de containers em comparação com o *hypervisor* quando o quesito é tempo de inicialização dos serviços.

O Apêndice A oferece informações sobre os comandos e *scripts* utilizados para a implementação e testes da solução, além de mais detalhes sobre a instalação e o manuseio de containers no Docker.

6 CONSIDERAÇÕES FINAIS

Nesse trabalho foi apresentado os conceitos sobre informação e sua importância, uma breve história da evolução da tecnologia da informação, o funcionamento dos computadores, processos, redes e sistemas distribuídos, virtualização e isolamento de recursos. Além disso, foi abordado conceitos sobre tolerância à falhas, alta disponibilidade e a íntima ligação com a segurança da informação.

Este trabalho apresentou uma solução em que a disponibilidade e a integridade das informações são protegidas, aumentando sensivelmente a segurança das informações contidas no sistema. É de se ressaltar que a tecnologia de containers adotada no trabalho, provê o isolamento de recursos, situação que também favorece a segurança dos dados do sistema.

Para o gerenciamento dos containers Linux, optou-se pela ferramenta Docker que produz um ambiente multiplataforma em que os conteiners podem ser executados também em sistemas operacionais Windows e MacOS.

Foi possível observar que a virtualização é uma importante ferramenta para a utilização mais eficiente dos recursos de *hardware*, e, tendo em vista que quando se fala em alta disponibilidade geralmente se fala na necessidade de um certo grau de redundância de recursos, ela se mostra uma solução economicamente interessante.

Os containers em particular são um modelo de virtualização mais leve e portanto muito mais ágil, principalmente quando se deseja levantar ou baixar serviços em um computador. Em questão de um ou dois segundos um banco de dados PostgreSQL está disponível para conexão, enquanto em uma máquina virtual, hospedada no mesmo servidor físico precisou de 37 segundos para que ela pudesse responder a um ping na rede.

Nesse sentido, verificou-se que o sistema é eficaz para subir rapidamente pequenos serviços quando é necessário e a situação assim se configura. No caso da configuração de ambientes de alta disponibilidade, é fácil a verificação de uma paralisação indesejada de serviço por problemas que ocorram, seja de *hardware* ou de *software*, sendo que providências podem ser automatizadas para o reestabelecimento dos serviços quase que de maneira instantânea, a depender apenas da disponibilidade dos dados persistidos, no caso em que se aplica.

Observou-se também que o sistema de containers tem um modelo de persistência peculiar (e limitado), sendo mais difícil de ser portado entre diferentes plataformas. Foi encontrado dificuldade na tentativa de implantação do sistema DRBD que faz espelhamento de discos através da rede, pois não foi possível encontrar uma solução para a montagem do dispositivo de blocos dentro do container. Também houve dificuldade para executar o container do PostgreSQL dentro do sistema operacional Microsoft Windows, uma vez que não se conseguiu dar permissões para o usuário "postgres" interno do container para acesso ao conteúdo dos dados do banco de dados, armazenados fora do container. Apesar do "bind" de montagem do Docker apresentar uma visão dos arquivos do diretório do Windows dentro do container, estes eram vistos como propriedades apenas do usuário root, não sendo possível então a utilização pelo SGBD.

Apesar de algumas dificuldades, percebeu-se que a tecnologia é eficiente e promissora, sendo que conhecendo bem, é possível se desenvolver muitos tipos de ferramenta de segurança através deste recurso.

A rapidez com que se consegue subir ou parar serviços isolados através do container, sugere que, em caso de uma rede com armazenamento externo disponível (um *storage* NAS ou um modelo baseado em iSCSI) é possível detectar o mal funcionamento de um container automaticamente, utilizando-se de *scripts* que observam através da rede, e então subir o mesmo container em um servidor remoto ou mesmo em um computador de mesa, de maneira quase instantânea, tendo assim serviços que podem subir em diferentes computadores físicos a qualquer momento, sempre que for necessário, aumentando sensivelmente a disponibilidade dos sistemas.

A integração do sistema de cluster com RAID nível 1 por *software*, seja por discos externos de interface USB, sejam por discos com interface SATA (com as devidas adaptações) mostrou-se uma forma eficiente e econômica de recursos, que oferece uma recuperação de desastres muito satisfatória para uma empresa de pequeno porte.

Uma sugestão para trabalhos futuros pode ser no sentido de se investigar o funcionamento do processo init, e como um processo init se integraria dentro de um container.

Na linha da alta disponibilidade, pode-se investigar também a integração do funcionamento do sistema de *cluster* DRBD para se criar um RAID nível 1 através da rede, ou seja, o espelhamento dos dados passa a ser feito em um local remoto, via rede. A integração desse tipo de serviço com a inicialização rápida do container de SGBD em caso de falhas, pode ser uma outra solução interessante.

Mais uma sugestão de trabalho futuro seria de configurar um ambiente de armazenamento para container baseado em arquivo interno montado como dispositivo de bloco, sendo que esta técnica pode aumentar a portabilidade das características de persistência do sistema de containers.

Outra sugestão para trabalhos futuros seria de configurar o sistema de *cluster* do próprio SGBD (PostgreSQL ou outro) que funcionaria dentro do container, possivelmente com os dados montados em um arquivo localizado em um diretório do sistema hospedeiro, enxergado internamente pelo container através de uma bind de montagem, e posteriormente, montado como dispositivo de blocos em um diretório interno onde ficam os dados do SGBD. Assim, o sistema passaria a ser portável para qualquer ambiente, tornando o sistema mais flexível e resiliente.

REFERÊNCIAS

ALVIM, Fernanda Cristina da Silva. A gestão da tecnologia da informação (TI) nas micros e pequenas empresas. In: **Revista eletrônica fundação educacional São José**, 3a Ed. Santos Dumont, 6 Set. 2015. Disponível em:<http://fsd.edu.br/revistaeletronica/arquivos/3Edicao/artigo22%20FERNANDA.pdf>. Acesso em 20 mai. 2017.

AMD. **AMD PRO Série A Virtualização do cliente**. [s.i.], 2017. Disponível em: <http://www.amd.com/pt-br/solutions/pro/virtualization>. Acesso em 2 Nov. 2017.

ASSOCIAÇÃO BRASILEIRA DE NORMAS TÉCNICAS - ABNT. **NBR ISO/IEC 27001:2013**: tecnologia da informação: técnicas de segurança: sistema de gestão da segurança da informação: requisitos. Rio de Janeiro: ABNT, 2013. 30p.

AWS. **O que é computação em nuvem**? [s.i.]: Amazon Web Services Inc., 2017. Disponível em: <https://aws.amazon.com/pt/what-is-cloud-computing/>. Acesso em: 19 Nov. 2017.

AZURE, Microsoft. **O que é computação em nuvem**? Um guia para iniciantes. [s.i.]: Microsoft Azure, 2017. Disponível em:<https://azure.microsoft.com/pt-br/overview/what-is-cloud-computing/>. Acesso em 19 Nov. 2017.

BARONI, Rodrigo Ferreira. **Configurando user mode Linux [UML] e um ambiente para desenvolvimento e depuração do kernel do Linux**. Instituto de matemática e estatística da Universidade de São Paulo., 2007. Disponível em: <https://www.ime.usp.br/~baroni/docs/uml.html>. Acesso em 19 set. 2017.

BSD.ORG. ***Welcome to www.bsd.org***!.[S.I.]:bsd.org, 2011. Disponível em: <http://www.bsd.org/>. Acesso em 5 Nov. 2017.

CACIATO, Luciano Eduardo. Virtualização e Consolidação dos servidores do Datacenter. **Centro de Computação da Universidade Estadual de Campinas–São Paulo.** Campinas, 2009. Disponível em: <http://www.ccuec.unicamp.br/biti/download/Artigo_Virtualizacao_Datacenter.Pdf> Acesso em 9 Abr. 2017.

CANONICAL. ***LinuxContainers.org infrastructure for container projects***. [S.I.]: Canonical Ltd, [s.d.]. Disponível em: <https://linuxcontainers.org>. Acesso em 23 Out. 2017.

CANONICAL. ***LinuxContainers.org infrastructure for container projects***. [S.I.]: Canonical Ltd, [s.d.]b. Disponível em: <https://linuxcontainers.org/lxc/security/>. Acesso em 19 Out. 2017.

CARUSO, A. A. Carlos.; STEFFEN, Flávio Deny. **Segurança em informática e de informações**. 2a Ed. São Paulo: SENAC, 1999, 367p.

CARVALHO, Flávio. O mercado de segurança da informação no Brasil. **Jornal do Brasil**, Rio de Janeiro, 19 mar. 2013. País – Sociedade aberta. Disponível em:

<http://www.jb.com.br/sociedade-aberta/noticias/2013/03/19/o-mercado-de-seguranca-da-informacao-no-brasil/> Acesso em 21 Mai. 2017.

DEBIAN. **O manual do administrador Debian**. [s.d.] Disponível em: <https://debian-handbook.info/browse/pt-BR/stable/sect.apparmor.html>. Acesso em: 10 Nov. 2017.

DEFLEUR, Melvin L.; BALL-ROKEACH, Sandra. **Teorias da comunicação de massa**. 1ª Ed. Rio de Janeiro: Jorge Zahar Ed, 1993, 397p.

DIE.NET. ***Linux Man Pages***. Linux Documentation, [s.d.]. Disponível em: <https://linux.die.net/man/>. Acesso em: 25 Ago. 2017.

DOCKER. ***What is Docker***? [S.I.]: Docker Inc., 2017. Disponível em: <https://www.docker.com/what-docker>. Acesso em 25 Out. 2017.

DOCKER. ***What is a container***? [S.I.]: Docker Inc., 2017b. Disponível em: <https://www.docker.com/what-container>. Acesso em 25 Out. 2017.

DOCKER. ***Docker for Windows.*** [S.I.]: Docker Inc., 2017c. Disponível em: <https://www.docker.com/docker-windows>. Acesso em 16 Nov. 2017.

DOCKER. ***Docker for MAC.*** [S.I.]: Docker Inc., 2017d. Disponível em: <https://www.docker.com/docker-mac>. Acesso em 16 Nov. 2017.

FERREIRA, Aurélio Buarque de Holanda. **Dicionário Aurélio da língua portuguesa**. 5a Ed. Curitiba:Positivo, 2010, 2272p.

FREE SOFTWARE FOUNDATION. ***GNU Coreutils.*** [S.I.]: Free Software Foundation, 2017. Disponível em: <http://www.gnu.org/software/coreutils/manual/html_node/chroot-invocation.html#chroot-invocation>. Acesso em 30 Out. 2017.

FULAY, Adeesh. ***Containers Deep Dive – LXC vs Docker***. San Jose: Robin Systems Inc., 2017. Disponível em:<https://robinsystems.com/blog/containers-deep-dive-lxc-vs-docker-comparison/>. Acesso em: 18 Nov. 2017.

FUSSELL, Mark. **Por que usar uma abordagem de microsserviço para construir aplicativos**? [s.i]: Microsoft Azure, 2017. Disponível em: <https://docs.microsoft.com/pt-br/azure/service-fabric/service-fabric-overview-microservices>. Acesso em: 18 Nov. 2017.

GOLDBERG, Ian *et al*. ***A secure environment for untrusted helper applications****: Confining the Wily Hacker*. Berkley: University of California, Computer Science Division, 1996, 14p. Originally published in the Proceedings of the Sixth USENIX UNIX Security Symposium. Disponível em: <https://www.usenix.org/legacy/publications/library/proceedings/sec96/full_papers/goldberg/goldberg.pdf>. Acesso em 7 de Out. 2017.

GRATTAFIORI, Aaron. ***Understanding and hardening Linux Containers***. Version 1.1. [S.I]: NCC Group, 2016, 123p. Disponível em: <https://www.nccgroup.trust/

globalassets/our-research/us/whitepapers/2016/april/ncc_group_understanding_hardening_linux_containers-1-1.pdf>. Acesso em 9 Out. 2017.

GUIMARÃES, Angelo de Moura; LAGES, Alberto de Castilho. **Introdução à ciência da computação**. 1a Ed. Rio de Janeiro: Livros Técnicos e Científicos, 1985, 165p.

HESS, Pablo. **Bê-á-bá do MAC no Linux**. [S.I.]: IBM developerWorks, 2010. Disponível em: <https://www.ibm.com/developerworks/community/blogs/752a690f-8e93-4948-b7a3-c060117e8665/entry/b_c3_aa__c3_a1_b_c3_a1_do_mac_no_linux1?lang=en>. Acesso em: 9 Nov. 2017.

INTEL. ***Intel virtualization technology***. [s.i.]., [s.d]. Disponível em:<https://www.intel.com/content/www/us/en/virtualization/virtualization-technology/intel-virtualization-technology.html>. Acesso em: 2 Nov. 2017.

KERNEL. **linux/fs/namespace.c**. [s.i.]: kernel.org, 2017. Kernel v. 4.13.11. Disponível em: <https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux-stable.git/tree/fs/namespace.c?h=v4.13.11>. Acesso em 25 Nov. 2017.

KERNEL. **linux/fs/open.c**. [s.i.]: kernel.org, 2017b. Kernel v. 4.13.11. Disponível em: <https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux-stable.git/tree/fs/open.c?h=v4.13.11>. Acesso em 25 Nov. 2017.

KERRISK, Michael. ***Namespaces in operation, part 1****: namespaces overview*. [s.i.]: LWN.net, 4 Jan. 2013. Disponível em: <https://lwn.net/Articles/531114/>. Acesso em 4 Nov. 2017.

KIVITY, Avi *et al*. ***kvm****: the Linux Virtual Machine Monitor*. Ottawa: Linux Symposium, 2007, Volume One: p. 225–230. Disponível em: <https://www.kernel.org/doc/ols/2007/ols2007v1-pages-225-230.pdf>. Acesso em: 6 Out. 2017.

KUBERNETES. ***Production-grade container orchestration***. [s.i.]: The Linux Foundation, 2017. Disponível em: <https://kubernetes.io/>. Acesso em 19 Nov. 2017.

KVM. ***Kernel Virtual Machine***. [s.d.] Disponível em: <https://www.linux-kvm.org/page/Main_Page> Acesso em 2 de Out. 2017.

LINDHOLM, Tim *et al*. ***The Java ® Virtual Machine specification****: Java SE 8 Edition*. Redwood City: Oracle America, 2015, 590p. Disponível em: <https://docs.oracle.com/ javase/specs/jvms/se8/jvms8.pdf>. Acesso em 9 Out. 2017.

LINUX KERNEL ORGANIZATION. ***About Linux kernel****: What is Linux?*. [S.I.]:Linux Kernel Organization, Inc., 2017. Disponível em: <https://www.kernel.org/linux.html>. Acesso em 5 Nov. 2017.

LINUX KERNEL ORGANIZATION. ***Good for you, you've decided to clean the elevator!*** [S.I.]:Linux Kernel Organization, Inc., 2017b. Smack. Disponível em:

<https://www.kernel.org/doc/Documentation/security/Smack.txt>. Acesso em 26 Nov. 2017.

LINUX KERNEL ORGANIZATION. ***What is Tomoyo****?* [S.I.]:Linux Kernel Organization, Inc., 2017c. Tomoyo. Disponível em: <https://www.kernel.org/doc/Documentation/security/tomoyo.txt>. Acesso em 26 Nov. 2017.

LINUX KERNEL ORGANIZATION. ***Secure computing with filters****!* [S.I.]:Linux Kernel Organization, Inc., 2017d. Seccomp. Disponível em: <https://www.kernel.org/doc/Documentation/prctl/seccomp_filter.txt>. Acesso em 26 Nov. 2017.

LINUX MAN-PAGES PROJECT. ***Linux man pages online***. Munich: man7.org Training and Consulting, 2017. Disponível em: <http://man7.org/linux/man-pages/index.html>. Acesso em 30 Out. 2017.

LOPES FILHO, Edmo; **Arquitetura de alta disponibilidade para firewall e IPS baseada em SCTP**. Uberlândia: Universidade Federal de Uberlândia, 2008. Disponível em: <http://repositorio.ufu.br/bitstream/123456789/12466/1/Edmo.pdf> Acesso em 30 Ago. 2017.

MANZANO, André Luiz N.G.; MANZANO, Maria Isabel N.G. **Informática básica**. 2a Ed. São Paulo: Érica, 1998, 178p.

MARTINS, Henrique Pachioni. Tolerância a falha em um ambiente de computação em nuvem open source. **Universidade Estadual Paulista “Júlio de Mesquita Filho”**. Bauru, 2014. Disponível em: <https://repositorio.unesp.br/bitstream/handle/11449/127619/000843909.pdf?sequence=1> Acesso em 12 Dez. 2017.

MASSIGLIA, P.; MARCUS, E. (Ed.). ***The Resilient Enterprise****: recovering information services from disasters*. Mountain View/CA, USA: Veritas software corporation, 2002, 527p.

MCCABE, John; FRIIS, Michael. ***Introduction to Windows containers***. Redmond: Microsoft Press, 2017, 70p. Disponível em: <https://aka.ms/containersebook>. Acesso em: 6 Out. 2017.

MICROSOFT. ***Windows containers****: what are containers*. [s.d.] Disponível em <https://docs.microsoft.com/en-us/virtualization/windowscontainers/about/>. Acesso em 14 Set. 2017.

MICROSOFT. ***Why Hyper-V?*** *Competitive Advantages of Windows Server 2012 Hyper-V over VMware vSphere 5.1*. Microsoft Corporation, 2012. Disponível em: <http://download.microsoft.com/download/5/8/2/58258EB6-39A9-4C44-8E58-328559A9DC78/Competitive-Advantages-of-Windows-Server-2012-Hyper-V-over-Vmware-vSphere-5-1-EN.pdf>. Acesso em: 8 Out. 2017.

MICROSOFT. **Windows**. [S.I.]: Microsoft, 2017. Disponível em: <https://www.microsoft.com/pt-br/windows>. Acesso em 5 Nov. 2017.

MIT. ***Massachusetts Institute of Technology***. [s.i.], [s.d.]. Disponível em: <http://web.mit.edu/>. Acesso em 2 Nov. 2017.

MOCK, Jim. ***FreeBSD handbook***. *Revision 51016*. [S.I.]:The FreeBSD Documentation Project, 2017, 721p. *Chapter 1.3.1.* ***A brief history of FreeBSD***: p. 9–10. Disponível em: <https://download.freebsd.org/ftp/doc/en/books/handbook/book.pdf>. Acesso em: 7 Out. 2017.

ORACLE. ***Oracle® Solaris administration****: Oracle Solaris Zones, Oracle Solaris 10 Zones, and Resource Management*. Redwood City: Oracle America, 2012, 428p. Part No: 821–1460–12. Disponível em: <https://docs.oracle.com/cd/E23824_01/pdf/821-1460.pdf>. Acesso em: 7 Out. 2017.

ORACLE. ***Oracle VM VirtualBox user manual***. [S.I.]: Oracle Corporation, 2017, 360p. Disponível em: <http://download.virtualbox.org/virtualbox/5.1.22/UserManual.pdf>. Acesso em: 8 Out. 2017.

PARALLELS. ***OpenVZ users guide***. Schaffhausen: Parallels IP Holdings GmbH, 2016, 119p. Disponível em: <https://docs.openvz.org/openvz_users_guide.pdf>. Acesso em: 8 Out. 2017.

PARALLELS. ***Virtuozzo 7 users guide***. Schaffhausen: Parallels IP Holdings GmbH, 2017, 197p. Disponível em: <https://docs.virtuozzo.com/pdf/virtuozzo_7_users_guide.pdf>. Acesso em: 8 Out. 2017.

PAVANI, Luciana. Mercado de TI deve crescer 3% em 2016 no Brasil. **Estadão**, São Paulo, 06 jun. 2016. Link. Disponível em: <http://link.estadao.com.br/noticias/empresas, mercado-de-ti-deve-crescer-3-em-2016-no-brasil,10000055519>. Acesso em 18 Jun. 2017.

POLIZELLI, Demerval L. *et al*. **Sociedade da informação**: os desafios da era da colaboração e da gestão do conhecimento. São Paulo: Saraiva, 2008, 259p.

PRATES, Glaúcia Aparecida; OSPINA, Marco Túlio. Tecnologia da informação em pequenas empresas: fatores de êxito, restrições e benefícios. In: **Revista de Administração Contemporânea**, v. 8, n. 2, p. 9-26, 2004. Disponível em: <http://www.scielo.br/pdf/rac/v8n2/v8n2a02.pdf> Acesso em 18 Mai. 2017.

QEMU. ***Qemu readme***. [s.d.] Disponível em:<https://github.com/qemu/qemu>. Acesso em 23 Set. 2017.

RAFAEL, Gustavo de Castro. A realidade de ambientes de TI em micro e pequenas empresas (MPE). In: **PTI – Profissionais de TI**, 2014. Disponível em: <https://www.profissionaisti.com.br/ 2014/02/a-realidade-de-ambientes-de-ti-em-micro-e-pequenas-empresas-mpe/> Acesso em 20 Mai. 2017.

REDHAT. **O que é um container Linux**? [s.d.] Disponível em <https://www.redhat.com/pt-br/containers/whats-a-linux-container>. Acesso em 2 Set. 2017.

REDHAT. **Alcance uma arquitetura de microsserviços bem-sucedida**. [s.i.]: Red Hat Inc., 2016, 4p. Disponível em: https://www.redhat.com/cms/managed-files/mi-microservices-architecture-design-whitepaper-inc0336100lw-201602-a4-ptbr.pdf>. Acesso em 18 Out. 2017.

RIONDATO, Matteo. ***FreeBSD handbook**. Revision 51016*. [S.l.]:The FreeBSD Documentation Project, 2017, 721p. *Chapter 14. **Jails***: p. 263–278. Disponível em: <https://download.freebsd.org/ftp/doc/en/books/handbook/book.pdf>. Acesso em: 7 Out. 2017.

SEO, Kyoung-Taek *et al*. *Performance comparison analysis of linux container and virtual machine for building cloud*. **Advanced Science and Technology Letters**, Jeju, v. 66, n. 25, p. 105–111, 2014. SERSC. ISSN: 2287-1233 ASTL

SILBERSCHATZ, Abraham; GALVIN, Peter Baer. **Sistemas operacionais**: conceitos. 1ª Ed. São Paulo: Prentice Hall, 2000, 903p.

SILBERSCHATZ, Abraham; GALVIN, Peter Baer; GAGNE, Greg. **Fundamentos de sistemas operacionais**. 8ª Ed. Rio de Janeiro: LTC, 2010, 515p.

SOBELL, Mark G. ***Unix System V**: A practical guide*. 3rd Ed. United States: Pearson, 1995, 800p.

SORIMA NETO, João. Brasil ganha espaço em segurança da informação. **O Globo**, Rio de Janeiro, 27 abr. 2017. Economia. Disponível em:<http://oglobo.globo.com/economia/brasil-ganha-espaco-em-seguranca-da-informacao-17644315>. Acesso em 21 Mai. 2017.

STALLINGS, William. **Arquitetura e organização de computadores**. 8ª Ed. São Paulo: Pearson Pratice Hall, 2010, 624p.

STATO, André. **Desmistificando o *fork bomb***. [s.i.]: Blog Stato, 2016. Disponível em: <http://stato.blog.br/wordpress/desmestificando-o-fork-bomb/>. Acesso em 25 Nov. 2017.

SYMANTEC. **Glossário**: *DoS (denial-of-service) attack*. [s.i.]: Symantec corporation, 2017. Disponível em: <https://www.symantec.com/pt/br/security_response/glossary/define.jsp?letter=d&word=dos-denial-of-service-attack>. Acesso em 25 Nov. 2017.

TANENBAUM, Andrew S. **Redes de computadores**. 4a Ed. Rio de Janeiro: Elsevier, 2003, 945p.

TANENBAUM, Andrew S. **Sistemas operacionais modernos**. 2a Ed. São Paulo: Pearson Prentice Hall, 2003, 695p.

TANENBAUM, Andrew S. **Organização estruturada de computadores**. 5a Ed. São Paulo: Pearson, 2007, 449p.

TANENBAUM, Andrew S.; STEEN, Maarten van. **Sistemas distribuídos**: princípios e paradigmas. 2ª Ed. São Paulo: Pearson Pratice Hall, 2007, 382p.

THE LINUX FOUNDATION. ***Open Container Initiative****: About*. [S.I.]: The Linux Foundation, 2016. Disponível em:<https://www.opencontainers.org/about>. Acesso em 25 Out. 2017.

THE LINUX FOUNDATION. ***The Linux Foundation****: About*. [S.I.]: The Linux Foundation, 2017. Disponível em:<https://www.linuxfoundation.org/about/>. Acesso em 25 Out. 2017.

THE OPEN GROUP. **Unix**. [s.i.]: 2017. Disponível em: <http://www.opengroup.org/subjectareas/platform/unix>. Acesso em 4 Nov. 2017.

THIEL, Gregory et al. ***Adding individual database failover/switchover to an existing storage component with limited impact***. U.S. Patent n. 8,275,907, 25 set. 2012.

UOL. **Linux containers (LXC)**: como funciona e quais são suas vantagens. São Paulo: Universo Onlone S/A, 2017. Disponível em: <http://uolhost.uol.com.br/academia/noticias/2015/02/24/linux-containers-lxc-como-funciona-e-quais-sao-suas-vantagens.html#rmcl>. Acesso em: 19 Nov. 2017.

VERAS, Manoel; CARISSIMI, Alexandre. **Virtualização de servidores**. Rio de Janeiro: Escola Superior de Redes, 2015, 172p. Disponível em: <https://www.scribd.com/doc/50570155/Virtualizacao-de-Servidores#fullscreen&from_embed> Acesso em 31 Ago. 2017.

VIEIRA, Luiz. Segurança da Informação: mudanças de paradigma com o avanço da civilização. In: **Espírito Livre**, Ed. #003 Junho 2009, pp. 60–62. Disponível em: <http://revista.espiritolivre.org/pdf/Revista_EspiritoLivre_003.pdf> Acesso em 27 mar. 2017.

VMWARE, Inc. **VMware ESX e VMware ESXi**: Os hypervisores líderes do mercado com produção comprovada. São Paulo: Vmware, 2009. Disponível em: <https://www.vmware.com/files/br/pdf/products/VMW_09Q1_BRO_ESX_ESXi_BR_A4_P6_R2.pdf> Acesso em 7 Out. 2017.

VMWARE, Inc. ***Using VMware Workstation***. Palo Alto: Vmware, 2011. Disponível em: <https://www.vmware.com/pdf/ws80-using.pdf> Acesso em 6 Out. 2017.

VSERVER, Linux. ***Linux Vserver***: Overview. [S.I.]: Linux Vserver, 2013. Disponível em:<http://linux-vserver.org/Overview>. Acesso em: 8 Out. 2017.

XEN PROJECT. ***About us***. The Linux Foundation, 2013. Disponível em: <https://www.xenproject.org/about-us.html>. Acesso em 1 de Out. 2017.

WOOD, Michael B. **Introdução à segurança do computador**. 1a Ed. Rio de Janeiro: Campus, 1984, 138p.

APÊNDICE A – INSTALAÇÃO DOS CONTAINERS

A instalação do aplicativo Docker no Linux Ubuntu é fácil.

No servidor físico central é utilizado a distribuição Ubuntu Server 16.04 LTS de 64 bits do sistema operacional Linux, sem nenhuma interface gráfica. Nos desktops é utilizado a distribuição Ubuntu 16.04 LTS para *desktop*, com interface gráfica Unit. No *site* do Docker, há a informação que a utilização de "*Docker for Ubuntu*", (ou Docker para Ubuntu) é a maneira mais fácil para implantação deste aplicativo nesta distribuição e que há uma versão comunitária gratuita (CE) e uma versão paga (com assinatura) que dá direito a suporte e certificação de que o container pode ser instalado em infraestruturas de *cloud* (nuvem) ou ainda em estruturas *bare metal* (direto no *hardware*).
https://www.docker.com/docker-ubuntu

Para o uso do Docker CE para plataforma Ubuntu, as arquiteturas suportadas são x86_64, armhf e s390x (sistemas z da IBM). Neste trabalho é utilizado a arquitetura x86_64.

Assim, foram criados duas instâncias de containers Docker. Uma instância refere-se a um container Docker montado com a imagem "postgre" para executar o serviço de SGBD, e a outra instância utilizando uma imagem "java:8", para a execução do servidor de aplicação, ambas baixadas do Docker Hub.

Primeiro foram criados os containers no servidor físico central. O Docker baixa via internet as imagens de containers do Postgresql e do Java automaticamente do Docker Hub.

Em princípio, foi criado um "alias" de endereçamento de rede virtual relativo a interface física de rede no sistema operacional, para cada um dos containers, para que estes pudessem ser acessados externamente, independente da máquina que estivessem funcionando, sempre pelo mesmo endereço IP:

```
#ifconfig enp4s0:1 192.168.0.25 netmask 255.255.255.0 up
#ifconfig enp4s0:2 192.168.0.26 netmask 255.255.255.0 up
```

A ideia é que estas mesmas configurações de endereçamento de rede sejam realizadas separadamente em dois computadores de mesa em caso de desastre, para que em cada computador exista um ambiente com o mesmo endereço IP disponível para o container que nele estiver hospedado.

Para a instalação do sistema PostgreSQL, primeiramente foram instalados dois discos rígidos externos, portáteis de interface USB 3.0. Optou-se por configurá-los em um sistema de RAID nível 1 por *software*, ou seja, um tipo de espelhamento.

Isso significa que estes discos foram configurados para serem enxergados como um único dispositivo de bloco de dados, ou seja, um dispositivo único de memória secundária enxergado em um diretório /dev/md0.

```
#mdadm --create /dev/md0 --level=1 --raid-devices=2 missing
/dev/sdb1
#mdadm /dev/md0 -a /dev/sdc1
#mke2fs -t ext4 /dev/md0
```

O dispositivo /dev/md0, após criado o sistema de arquivos ext4, foi montado na pasta /home/cluster/no0.

```
#mount /dev/md0 /home/cluster/no0/
```

Na pasta /home/cluster/no0, alguns subdiretórios foram criados:

- postgresql
- appserver
- fserver (para uma implementação futura de um servidor de arquivos)

A pasta /home/cluster/no0/postgresql, por sua vez, foi montada internamente no container "db" apontando diretamente para a pasta onde ficam tradicionalmente armazenados os dados do SGBD PostgreSQL, através de um "bind" (a replicação da visibilidade de um diretório do sistema operacional hospedeiro em um diretório interno do container):

```
#docker run --name db -d -p 192.168.0.25:5432:5432 --user "$
(id -u):$(id -g)" -v
/home/cluster/no0/postgresql:/var/lib/postgresql/data -e
POSTGRES_PASSWORD=147258 postgres
```

De outro lado, a pasta /home/cluster/no0/serverapps foi montada internamente no container "as" apontando diretamente para a pasta /home/user/apps, que também é o diretório onde o comando de inicialização do servidor de aplicação seria executado:

```
#docker run --name as -d -p 192.168.0.26:5000:5000 --user "$
(id -u):$(id -g)" -v
/home/cluster/no0/serverapps:/home/user/apps -w
/home/user/apps java:8 java -jar AppServer.jar
```

Passou a existir dados do banco de dados PostgreSQL replicados em ambos os discos externos que, por terem interface USB, podem ser fisicamente levados para os terminais com grande facilidade em casos de desastre.

Foi criado um *script* que, com auxílio da ferramenta cron (um agendador de tarefas) do Linux, executado no servidor *host* central, verifica a cada cinco segundos se os processos referentes ao container de SGBD, bem como ao container de servidor de arquivos, estão em execução no sistema operacional.

Para checar se o processo do SGBD está em execução, o *script* primeiramente verifica se realmente existe uma instrução de paralisação no diretório cluster:

```
if [ ! -f /home/cluster/stop ]; then
```

Convencionou-se para esta solução, que a permissão para que um container possa ser paralisado será dada através da criação de um arquivo de nome “stop” criado dentro da pasta /home/cluster.

Após verificar que a paralisação de um container não é permitida, o *script* segue verificando o PID do processo e de acordo com o retorno toma a providência necessária.

O comando para verificar o PID do processo do SGBD é:

```
pidProcess=$(ps ax | grep postgres | grep -v grep | awk
'{print$1}')
```

No caso do processo relativo ao container de servidor de aplicação, o comando é semelhante:

```
pidProcess=$(ps ax | grep AppServer.jar  | grep -v grep | awk
'{print$1}')
```

Caso o resultado da consulta identifique que um dos processos não está em funcionamento, o mesmo é instigado a funcionar novamente:

```
docker start db
```

No caso do servidor de aplicação:

```
docker start as
```

O *script* tenta subir cada serviço por três vezes consecutivas, fazendo um registro de Log em cada tentativa, mesmo que bem sucedida. Este tipo de estratégia ajuda a manter o serviço disponível em casos de falhas de *software*. Uma simulação de falha, foi feita desligando-se o serviço de SGBD com um comando kill, enquanto um programa cliente realizava uma consulta:

```
kill -9 5249
```

Tal simulação fez com que o servidor de aplicação perdesse a conexão com o banco de dados, gerando uma linha de exceção no arquivo de *log* do aplicativo. Entretanto, o *script* recolocou o container de SGBD no ar quase instantaneamente,

sendo que o programa cliente não percebeu o problema, senão por um intervalo de quatro segundos, que foi o tempo necessário para que o resultado da consulta aparecesse na tela.

Em caso de as tentativas de ativação dos processos não surtirem resultados, o *script* encaminha um e-mail para um funcionário treinado e para a equipe de informática avisando do problema com o *software*.

A partir disso, há um outro *script* que pode ser acionado, opcionalmente também por intervenção de um usuário, que tenta então inicializar outra instância de container idêntica a aquela que não funcionou, todavia esta instância pega os dados de um outro diretório (reserva) contido em outro disco rígido do servidor central:

```
#docker run --name db_reserva -d -p 192.168.0.25:5432:5432
--user "$(id -u):$(id -g)" -v
/reserva/no0/postgresql:/var/lib/postgresql/data -e
POSTGRES_PASSWORD=147258 postgres
```

Isto também ocorre no caso do servidor de aplicação:

```
#docker run --name as_reserva -d -p 192.168.0.26:5000:5000
--user "$(id -u):$(id -g)" -v
/reserva/no0/serverapps:/home/user/apps -w /home/user/apps
java:8 java -jar AppServer.jar
```

Também foi criado um *script* que realiza a parada dos containers e em seguida realiza uma cópia dos dados para um local separado (reserva), dentro do próprio servidor. Este *script* toma o cuidado de paralisar o aplicativo cron que mantém os serviços ligados antes de prosseguir com a tarefa. Ele é executado uma vez por dia, sendo uma às 0h00, pois neste horário há pouca ou nenhuma utilização do sistema das lojas. A cópia é sincronizada com o aplicativo *rsync* em poucos minutos e em seguida, o *script* coloca o container novamente em funcionamento:

```
#touch /home/cluster/stop
#docker stop db
#docker stop as
#nohup rsync -Cravzp /home/cluster/no0/* /reserva/no0/ &
#docker start db
#docker start as
#rm /home/cluster/stop
```

No primeiro comando, o sistema cria um arquivo de nome “stop” dentro do caminho /home/cluster, sinalizando para o sistema que o *script* está paralisando o sistema propositalmente. Posterior a isso, o segundo e o terceiro comandos param

os containers de SGBD e de servidor de aplicação, respectivamente. Em seguida o comando rsync é executado no intuito de sincronizar as cópias entre o diretório utilizado em produção com um diretório de reserva, localizado em outro disco rígido. Logo após a sincronização, o sistema sobe novamente os dois containers para, por fim, apagar o arquivo "stop", sinalizando que o sistema deve funcionar normalmente.

Ao chegarem ao trabalho, os funcionários ligam os computadores de mesa que automaticamente executam outros *scripts* com a finalidade de atualizarem, agora via rede, suas máquinas com os dados do servidor. A atualização remota é feita também com o comando rsync, que trafega as alterações dos arquivos via rede do servidor central para o computador de mesa, o que também ocorre de maneira muito rápida.

```
#nohup rsync -Cravzp copia@192.168.0.246/reserva/no0/*
/reserva/no1/ &
```

Toda a comunicação do rsync que corre através da rede é realizada através do protocolo ssh, sendo que a autenticação da conexão ocorre através de chaves de criptografia trocadas previamente entre os servidores.

Em caso de desastre, optou-se por levantar os serviços manualmente, assim que o problema for detectado por algum usuário da empresa. Foi criado um procedimento para o reestabelecimento do sistema em caso de pane, sendo que o levantamento dos serviços poderá ser realizado com base nos discos externos (recomendado) ou, em último caso, com base nos dados de reserva realizada via rede.

Caso seja percebida a ausência de acesso ao sistema, verificando-se que o servidor encontra-se inacessível, um procedimento de reinicialização do servidor está descrito como padrão. Em caso de não haver resposta por parte do servidor, um procedimento está descrito para que o usuário, devidamente treinado, pegue um dos discos rígidos externos do servidor e ligue apenas um deles ao computador de mesa. Após, o usuário deve executar um *script* especial que verifica a presença do disco, detecta-os e os configura para executar novamente os containers.

Esse script tem o intuito de fazer o sistema estar disponível em regime de contingência.

Basicamente os comandos mais importantes utilizados dentro do *script* para levantar o serviço no computador de mesa do usuário são os seguintes:

```
#mdadm --assemble --scan
#mount /dev/md/0 /home/cluster/no1/
#ifconfig eth0:1 192.168.0.25 netmask 255.255.255.0 up
```

```
#docker start db
```

O primeiro comando detecta o disco configurado em RAID 1 e o insere no sistema. Já o segundo, monta o dispositivo de bloco em um diretório utilizado pelo container. Por fim, o container é executado colocando o serviço em funcionamento.

Assim, o SGBD estará funcionando. Caso não consiga obter sucesso com este disco, o usuário pode utilizar o outro que é uma cópia idêntica, tentando executar o mesmo *script*.

Outro computador de mesa dividirá a tarefa executando o servidor de aplicação, sendo que nele um outro *script* será executado, inciando os serviços à partir da cópia reserva do computador. Isso é possível porquê o servidor de aplicação não precisa ter uma cópia muito atualizada, como é o caso do SGBD. Ao inicializar o outro *script* no segundo computador, este irá executar alguns comandos, tendo os seguintes como principais:

```
#ifconfig eth0:1 192.168.0.26 netmask 255.255.255.0 up
#docker start as_reserva
```

O primeiro comando, estabelece um alias com o endereçamento IP correto do serviço para ser utilizado pelo container de servidor de aplicação, sendo que o segundo comando inicializa o container em si.

Desta forma, todos os serviços se estabelecem, mesmo que o servidor físico principal venha a falhar e não se reestabeleça.

Foi realizado um teste comparativo, utilizando-se uma máquina virtual contendo o sistema operacional Ubuntu 16.04 LTS Server básico, hospedado no mesmo computador físico servidor onde encontram-se os containers. Um *script* de teste foi feito para subir a máquina virtual e começar o processo de Ping no endereço IP da máquina virtual. Os pings não obtiveram resposta durante aproximadamente 37 segundos, até que a VM estivesse com o sistema de rede funcionando e pudesse respondê-los.

De outro lado, foi construído um outro *script* para que, em conjunto com uma aplicação Java que faz conexões JDBC ao SGBD no container, pudesse monitorar quanto tempo o SGBD levaria para aceitar a primeira conexão. Desde que o comando para subir o container foi disparado, o tempo até a primeira conexão ser aceita foi inferior a dois segundos, deixando claro a enorme vantagem do sistema de containers em comparação com o *hypervisor* quando o quesito é tempo de inicialização dos serviços.