Universidad de San Carlos de Guatemala
Facultad de Ingeniería
Escuela de Ingeniería en Ciencias y Sistemas

IMPLEMENTACIÓN DE ARQUITECTURA DE MICROSERVICIOS UTILIZANDO VIRTUALIZACIÓN POR SISTEMA OPERATIVO

William Estuardo Salazar Hernández

Asesorado por el Ing. Sergio Arnaldo Méndez Aguilar

Guatemala, febrero de 2017

UNIVERSIDAD DE SAN CARLOS DE GUATEMALA

FACULTAD DE INGENIERÍA

IMPLEMENTACIÓN DE ARQUITECTURA DE MICROSERVICIOS UTILIZANDO VIRTUALIZACIÓN POR SISTEMA OPERATIVO

TRABAJO DE GRADUACIÓN

PRESENTADO A LA JUNTA DIRECTIVA DE LA
FACULTAD DE INGENIERÍA
POR

WILLIAM ESTUARDO SALAZAR HERNÁNDEZ

ASESORADO POR EL ING. SERGIO ARNALDO MÉNDEZ AGUILAR

AL CONFERÍRSELE EL TÍTULO DE

INGENIERO EN CIENCIAS Y SISTEMAS

GUATEMALA, FEBRERO DE 2017

UNIVERSIDAD DE SAN CARLOS DE GUATEMALA

FACULTAD DE INGENIERÍA

NÓMINA DE JUNTA DIRECTIVA

DECANO	Ing. Pedro Antonio Aguilar Polanco
VOCAL I	Ing. Angel Roberto Sic García
VOCAL II	Ing. Pablo Christian de León Rodríguez
VOCAL III	Inga. Elvia Miriam Ruballos Samayoa
VOCAL IV	Br. Jurgen Andoni Ramírez Ramírez
VOCAL V	Br. Oscar Humberto Galicia Nuñez
SECRETARIA	Inga. Lesbia Magalí Herrera López

TRIBUNAL QUE PRACTICÓ EL EXAMEN GENERAL PRIVADO

DECANO	Ing. Pedro Antonio Aguilar Polanco
EXAMINADOR	Ing. Edgar Estuardo Santos Sutuj
EXAMINADOR	Ing. Miguel Ángel Cancinos Rendón
EXAMINADOR	Ing. Oscar Alejandro Paz Campos
SECRETARIA	Inga. Lesbia Magalí Herrera López

HONORABLE TRIBUNAL EXAMINADOR

En cumplimiento con los preceptos que establece la ley de la Universidad de San Carlos de Guatemala, presento a su consideración mi trabajo de graduación titulado:

IMPLEMENTACIÓN DE ARQUITECTURA DE MICROSERVICIOS UTILIZANDO VIRTUALIZACIÓN POR SISTEMA OPERATIVO

Tema que me fuera asignado por la Dirección de la Escuela de Ingeniería en Ciencias y Sistemas, con fecha abril de 2016.

William Estuardo Salazar Hernández

Universidad de San Carlos de Guatemala
Facultad de Ingeniería
Escuela de Ciencias y Sistemas

Guatemala, 06 de noviembre de 2016

Ingeniero
Carlos Alfredo Azurdia Morales
Coordinador del Área de Trabajos de Graduación

Respetable ingeniero Azurdia:

Por este medio le informo que, como asesor del trabajo de graduación del estudiante universitario de la carrera de Ingeniería en Ciencias y Sistemas, **WILLIAM ESTUARDO SALAZAR HERNÁNDEZ**, carné **201114539**, que he revisado el trabajo de graduación titulado: **"IMPLEMENTACIÓN DE ARQUITECTURA DE MICROSERVICIOS UTILIZANDO VIRTUALIZACIÓN POR SISTEMA OPERATIVO"**, y a mi criterio el mismo está completo y cumple con los objetivos propuestos para su desarrollo según el protocolo.

Agradeciendo su atención a la presente,

Atentamente,

Sergio Arnaldo Méndez Aguilar
Ingeniero en Ciencias y Sistemas
Colegiado No. 10958

Ing. Sergio Arnaldo Méndez Aguilar
Escuela de Ciencias y Sistemas
Asesor de trabajo de graduación
Colegiado: 10958

Universidad San Carlos de Guatemala
Facultad de Ingeniería
Escuela de Ingeniería en Ciencias y Sistemas

Guatemala, 23 de Noviembre de 2016

Ingeniero
Marlon Antonio Pérez Türk
Director de la Escuela de Ingeniería
En Ciencias y Sistemas

Respetable Ingeniero Pérez:

Por este medio hago de su conocimiento que he revisado el trabajo de graduación del estudiante **WILLIAM ESTUARDO SALAZAR HERNÁNDEZ** con carné **201114539**, titulado: **"IMPLEMENTACIÓN DE ARQUITECTURA DE MICROSERVICIOS UTILIZANDO VIRTUALIZACIÓN POR SISTEMA OPERATIVO"**, y a mi criterio el mismo cumple con los objetivos propuestos para su desarrollo, según el protocolo.

Al agradecer su atención a la presente, aprovecho la oportunidad para suscribirme,

Atentamente,

Ing. Carlos Alfredo Azurdia
Coordinador de Privados
y Revisión de Trabajos de Graduación

UNIVERSIDAD DE SAN CARLOS DE GUATEMALA
FACULTAD DE INGENIERIA EN CIENCIAS Y SISTEMAS
COORDINACION TESIS

ESCUELA DE INGENIERÍA EN CIENCIAS Y SISTEMAS

UNIVERSIDAD DE SAN CARLOS
DE GUATEMALA

FACULTAD DE INGENIERÌA
ESCUELA DE INGENIERÍA EN
CIENCIAS Y SISTEMAS
TEL: 24767644

El Director de la Escuela de Ingeniería en Ciencias y Sistemas de la Facultad de Ingeniería de la Universidad de San Carlos de Guatemala, luego de conocer el dictamen del asesor con el visto bueno del revisor y del Licenciado en Letras, del trabajo de graduación ***IMPLEMENTACIÓN DE ARQUITECTURA DE MICROSERVICIOS UTILIZANDO VIRTUALIZACIÓN POR SISTEMA OPERATIVO,*** *realizado por el estudiante WILLIAM ESTUARDO SALAZAR HERNÁNDEZ, aprueba el presente trabajo y solicita la autorización del mismo.*

"ID Y ENSEÑAD A TODOS"

UNIVERSIDAD DE SAN CARLOS DE GUATEMALA
DIRECCION DE INGENIERIA EN CIENCIAS Y SISTEMAS

Ing. Marlon Antonio Pérez Türk
Director
Escuela de Ingeniería en Ciencias y Sistemas

Guatemala, 06 de febrero de 2017

Universidad de San Carlos
de Guatemala

Facultad de Ingeniería
Decanato

DTG. 075.2017

El Decano de la Facultad de Ingeniería de la Universidad de San Carlos de Guatemala, luego de conocer la aprobación por parte del Director de la Escuela de Ingeniería en Ciencias y Sistemas, al Trabajo de Graduación titulado: **IMPLEMENTACIÓN DE ARQUITECTURA DE MICROSERVICIOS UTILIZANDO VIRTUALIZACIÓN POR SISTEMA OPERATIVO,** presentado por el estudiante universitario: **William Estuardo Salazar Hernández,** y después de haber culminado las revisiones previas bajo la responsabilidad de las instancias correspondientes, autoriza la impresión del mismo.

IMPRÍMASE:

Ing. Pedro Antonio Aguilar Polanco
Decano

Guatemala, febrero de 2017

UNIVERSIDAD DE SAN CARLOS DE GUATEMALA
DECANO
FACULTAD DE INGENIERIA

/gdech

ACTO QUE DEDICO A:

Dios	Por darme la vida, el entendimiento, las fuerzas y el consuelo que necesito a lo largo de toda mi vida.
Mis padres	Rodolfo Salazar y Ester Hernández, por su apoyo, su amor incondicional, su dedicación, su paciencia y esfuerzo para formarme con valores íntegros.
Mis hermanos	Claudia y Rodolfo Salazar, por brindarme su apoyo incondicional en los buenos y malos momentos, e inspirarme a seguir luchando por mis sueños.
Señorita	Ana María Espina León, mi querida novia y futura esposa, por todo su amor y apoyo, por ser mi inspiración para seguir adelante en todo momento.

AGRADECIMIENTOS A:

Dios Por derramar bendiciones sobre mi familia y siempre cuidar de mi salud, en todas las noches de arduo trabajo y desvelo que dio como resultado, mi formación profesional.

Mis padres Rodolfo Salazar y Ester Hernández de Salazar, por el apoyo y todo el esfuerzo que realizaron para poder ayudarme a culminar mis estudios.

Mis hermanos Claudia Salazar y Rodolfo Salazar, por brindarme todos los recursos necesarios en el transcurso de mi carrera y darme el apoyo moral para convertirme en un profesional.

Señorita Ana María Espina León, mi querida novia y futura esposa, por todo su apoyo para la culminación de mi carrera y por darme siempre ánimos de seguir adelante con todo su amor y paciencia.

Mis amigos Por todo el trabajo y esfuerzo que desempeñamos a lo largo de nuestra carrera para alcanzar nuestras metas, y por su desinteresada amistad.

Mi asesor	Sergio Méndez, por brindarme su conocimiento y experiencias a lo largo del desarrollo del mi trabajo de investigación.
La Universidad de San Carlos de Guatemala	Por formarme profesionalmente y ser la institución que educa a miles de guatemaltecos para poder llegar a ser un mejor país.
Facultad de Ingeniería	Por el alto nivel académico que me brindó y por futuras oportunidades que se me presentarán por ser egresado de esta Facultad.

ÍNDICE GENERAL

ÍNDICE DE ILUSTRACIONES

FIGURAS

TABLAS

LISTA DE SÍMBOLOS

Símbolo	Significado
$	Indicador de línea de comandos en modo usuario.
#	Indicador de línea de comandos en modo *root*.

GLOSARIO

Amazon	Es utilizado para describir la organización, compañía o empresa de nacionalidad estadounidense encargada del comercio electrónico y servicios de *cloud computing* a diferentes niveles.
Amazon Web Services	Es una colección de servicios de computación en la nube, ofrecidos por *Amazon*.
Bare-metal	Es otro tipo de nombre que se le da al hipervisor tipo 1.
Benchmark	Es una técnica utilizada para medir el rendimiento de un sistema o componente del mismo.
Commit	Se refiere a la idea de consignar un conjunto de cambios "tentativos" de forma permanente. Un uso popular es al final de una transacción de base de datos.
CPU	*Central Processing Unit*, es el hardware dentro de una computadora u otros dispositivos programables, que interpreta las instrucciones de un programa informático mediante la realización de las operaciones básicas aritméticas, lógicas y de entrada/salida del sistema.

Daemon	Es un tipo especial de proceso informático no interactivo, es decir, que se ejecuta en segundo plano en vez de ser controlado directamente por el usuario.
EXE	*Executable*. Es una extensión que se refiere a un archivo ejecutable de código reubicable, es decir que sus direcciones de memoria son relativas.
Framework	Define, en términos generales, un conjunto estandarizado de conceptos, prácticas y criterios para enfocar un tipo de problemática particular que sirve como referencia, para enfrentar y resolver nuevos problemas de índole similar.
Hardware	Conjunto de elementos físicos o materiales que constituyen una computadora o un sistema informático.
HTTP	*Hypertext Transfer Protocol*. Es el protocolo de comunicación que permite las transferencias de información en la *World Wide Web*.
IP	Protocolo de comunicación de datos digitales clasificado funcionalmente en la capa de red, según el modelo internacional *OSI*.

iSCSI	Es un estándar que permite el uso del protocolo *SCSI* sobre redes *TCP/IP*.
Kernel	Parte más importante de un sistema operativo, parte encargada de acceder a los distintos dispositivos de los que una computadora dispone.
KVM	*Kernel-based Virtual* Machine, es una solución para implementar virtualización completa con *Linux*.
Netflix	Es una empresa comercial estadounidense de entretenimiento que proporciona, mediante tarifa plana mensual *streaming* multimedia bajo demanda por Internet.
NFS	*Network File System* es un protocolo de nivel de aplicación, según el Modelo *OSI*, que es utilizado para sistemas de archivos distribuido en un entorno de red de computadoras de área local.
Nginx	Es un servidor *web/proxy* inverso, ligero, de alto rendimiento y un *proxy* para protocolos de correo electrónico.
Openstack	Es un proyecto de *cloud computing* para proporcionar una infraestructura como servicio.

PowerVM	Es un producto de *IBM* que permite la virtualización de capacidad de entornos *AIX*, *IBM i* y *Linux* sobre sistemas basados en procesadores *IBM POWER*.
RAM	*Random Access Memory*, se utiliza como memoria de trabajo de computadoras para el sistema operativo, los programas y la mayor parte del *software*.
Redis	Es un motor de base de datos en memoria, basado en el almacenamiento clave/valor.
Root	Es el nombre convencional de la cuenta de usuario que posee todos los derechos en todos los modos (mono o multi usuario).
SCRUM	Es un proceso en el que se aplican, de manera regular, un conjunto de buenas prácticas para trabajar colaborativamente, en equipo, y obtener el mejor resultado posible de un proyecto.
SMB/CIFS	*Server Message Block / Common Internet File System*. Protocolo de red que permite compartir archivos, impresoras, etcétera, entre nodos de una red de computadoras que usan el sistema operativo *Microsoft Windows*.
Software	Conjunto de programas y rutinas que permiten a la computadora realizar determinadas tareas.

Spring Cloud Config	Provee soporte del lado del cliente y del servidor, para externalizar la configuración en un sistema distribuido.
Spring Cloud Netflix	Es una plataforma que provee integración entre la plataforma de código abierto de *Netflix* y *Spring Boot*.
TI	Tecnología de la información. Es la aplicación de ordenadores y equipos de telecomunicación para almacenar, recuperar, transmitir y manipular datos con frecuencia, utilizado en el contexto de los negocios u otras empresas.
URL	*Uniform Resource Locator*. Es un identificador de recursos uniforme (*Uniform Resource Identifier*, *URI*) cuyos recursos referidos pueden cambiar, esto es, la dirección puede apuntar a recursos variables en el tiempo.
USB	*Universal Serial Bus* es un puerto que sirve para conectar periféricos a una computadora.
XP	*eXtreme Programming*. Es una metodología ágil centrada en potenciar las relaciones interpersonales como clave para el éxito en desarrollo de *software*.

RESUMEN

Este trabajo consiste en la investigación y práctica de los conceptos necesarios para crear una aplicación, utilizando una arquitectura de microservicios combinada con virtualización por sistema operativo para, posteriormente, desarrollar un análisis comparativo con una arquitectura monolítica para observar las ventajas y desventajas que conlleva trabajar con este enfoque.

El primer capítulo abarca los conceptos básicos necesarios para entender la terminología que se usará alrededor de los siguientes capítulos, como por ejemplo: los tipos de virtualización, y los tipos de arquitectura de software.

El segundo capítulo tiene conceptos relacionados con la estructura de la virtualización por sistema operativo, específicamente contenedores; entre estos se puede mencionar el concepto contenedor *Linux*, que es una imagen, el cliente y servidor *Docker*, etc.

El tercer capítulo tiene conceptos relacionados con la arquitectura de microservicios, específicamente se analizan las ventajas y desventajas de emplear dicha arquitectura, así como los principios que rigen este tipo de arquitectura.

Finalmente, en el cuarto capítulo, se realiza un análisis comparativo entre una aplicación con arquitectura monolítica y una, empleando una arquitectura de microservicios, utilizando virtualización por sistema operativo, dando énfasis en las ventajas y desventajas que conlleva implementar dicha arquitectura.

OBJETIVOS

General

Demostrar que la mejor opción para implementar el patrón de arquitectura de microservicios es con el uso de contenedores mediante, la realización de una aplicación para su posterior análisis comparativo, con una aplicación implementada con una arquitectura monolítica.

Específicos

1. Analizar el impacto en el tiempo de implementación que se tiene al usar una arquitectura de microservicios con virtualización por sistema operativo.

2. Comprender cómo la implementación de una arquitectura de microservicios con virtualización por sistema operativo, es el vértice para la unión de desarrollo ágil, entrega continua y *DevOps*.

3. Comparar los costos que implica el implementar una aplicación con arquitectura monolítica, y una utilizando la arquitectura de microservicios con virtualización por sistema operativo.

INTRODUCCIÓN

En la actualidad, las empresas dedicadas al desarrollo de aplicaciones se han encontrado con una serie de problemas dependiendo del tipo de arquitectura que elijan para realizar sus proyectos, tal es el caso de muchas empresas que desde sus inicios tomaron la decisión de elegir un enfoque de arquitectura monolítica, lo cual, tiempo después de que sus proyectos y equipos de trabajo se volvieran más grandes, se dieran cuenta que el tipo de arquitectura que eligieron en sus comienzos, repercute grandemente en cuestiones de tiempo de desarrollo e implementación; además de serios problemas a la hora de manejar fallos. Algunas empresas tomaron este ejemplo y se decidieron por tomar el enfoque de una arquitectura de microservicios para evitar los problemas del uso de una arquitectura monolítica, pero como todo tiene una contraparte, estas empresas se dieron cuenta que esta arquitectura también tiene sus desventajas, por ejemplo, el que la ocurrencia de un fallo desemboca en un efecto dominó en los servicios en la mayoría de ocasiones; además, es difícil tener un control de la unidad causante del problema. Una tecnología reciente promete evitar estos problemas: los contenedores, la cual aísla los procesos, haciendo muy simple el manejo y actualización de los servicios.

1. MARCO TEÓRICO

Este capítulo abarca los conceptos generales de virtualización, así como los conceptos generales de los patrones y estilos de arquitectura, necesarios para poder realizar la implementación de una arquitectura de microservicios combinada con virtualización por sistema operativo.

1.1. Virtualización

La empresa *VMware*, líder en este campo, define virtualización como “el proceso de crear virtual, en lugar de físicamente, la versión de algo”[1].

La virtualización se puede aplicar a servidores, sistemas operativos, dispositivos de almacenamiento, aplicaciones, o redes; además es la manera más eficaz de reducir costos de TI.

1.1.1. Máquina *host*

Sean Campbell, consultor por más de una década en *Microsoft* e *Intel*, define qué es una máquina *host* en su artículo, *An Introduction to Virtualization*, como “Una máquina anfitrión o *host* es la máquina física ejecutando el software de virtualización”[2].

[1] VMware, Inc. *Virtualization: How It Works*. Consulta: 30 de agosto de 2016.
[2] CAMPBELL, et. al. *Intel*. p. 2.

Una máquina *host* contiene los recursos físicos, tales como: memoria, espacio de disco duro, *CPU*, y otros, como acceso a red, que las máquinas virtuales utilizan.

1.1.2. Máquina virtual

De acuerdo con Campbell, una máquina virtual se define de la siguiente manera: “Un sistema informático virtual es también conocido como máquina virtual o *Virtual Machine (VM)*, es un contenedor de software perfectamente aislado, con un sistema operativo y aplicaciones en el interior”[3].

La definición anterior quiere decir que cada máquina virtual es completamente independiente y, por lo tanto, se pueden colocar múltiples máquinas virtuales en una única computadora, permitiendo ejecutar varios sistemas operativos y aplicaciones en un solo servidor físico, o *host*.

1.1.3. Software de virtualización

Un software de virtualización, según Campbel es “un término genérico que denota software que permite a un usuario ejecutar máquinas virtuales en una máquina *host”*[4].

Se pueden mencionar algunos programas de virtualización: *Virtual Box* de Oracle, *vSphere* de *VMware*, *KVM* para el *kernel* de *Linux*, *PowerVM* de IBM, *XenServer* de la empresa Citrix, QEMU gestionado por la comunidad *open source*, *Hyper-V* de Microsoft.

[3] CAMPBELL, et. al. *Intel*. p. 2.
[4] Ibíd.

1.1.4. Disco virtual

El término disco virtual tal como Campbell lo indica, se refiere a "la representación física de la máquina virtual en el disco de la máquina *host*. Un disco virtual se compone de un solo archivo o conjunto de archivos relacionados"[5].

En la máquina virtual, los discos virtuales aparecen como discos duros físicos. Estos tienen el beneficio de la portabilidad, ya que se pueden mover de una máquina física a otra, con un impacto limitado en los archivos.

1.1.5. Adiciones de máquina virtual

Campbell enuncia las adiciones de máquina virtual o *virtual machine additions* como "componentes que incrementan el rendimiento del sistema operativo *guest*"[6].

Las adiciones de máquina virtual proveen, por ejemplo, acceso a dispositivos *USB*, a dispositivos de *CD/DVD*; algunos mejoran la resolución de video, entre otros.

1.1.6. Carpetas compartidas

Una carpeta compartida es un recurso que sirve para que una máquina virtual pueda tener acceso a archivos, ya sea de otra máquina virtual o de la máquina *host*.

[5] CAMPBELL, et. al. *Intel*. p. 3.
[6] Ibid.

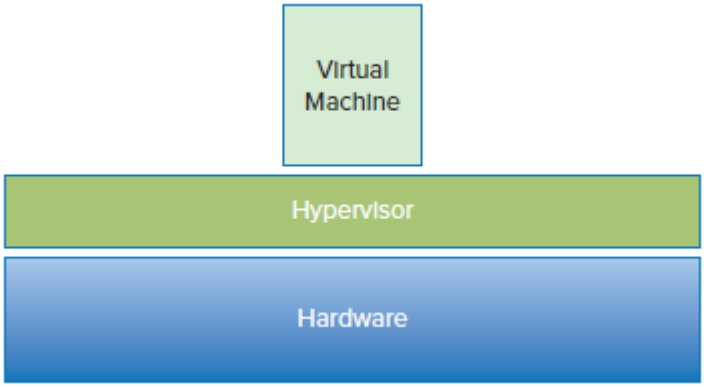
Normalmente se permite el uso de carpetas compartidas, tras la instalación de las adiciones de la máquina virtual.

1.1.7. Monitor de máquina virtual o hipervisor

Matthew Portnoy en su libro *Virtualization Essential*, define un hipervisor como “un software que maneja las interacciones entre cada máquina virtual y el hardware que todos los *guests* comparten”[7].

El hipervisor es una capa de software que reside debajo de las máquinas virtuales y encima del hardware. La figura siguiente ilustra dónde reside el hipervisor.

Figura 1. **Dónde reside el hipervisor**



Fuente: PORTNOY, Matthew. *Virtualization Essentials*. p. 20.

[7] PORTNOY, Matthew. *Virtualization Essentials*. p. 19.

Un sistema operativo se comunica directamente con el hardware debajo de el. Sin un hipervisor, las diferentes máquinas virtuales intentarían controlar simultáneamente el hardware, lo cual resultaría en un caos.

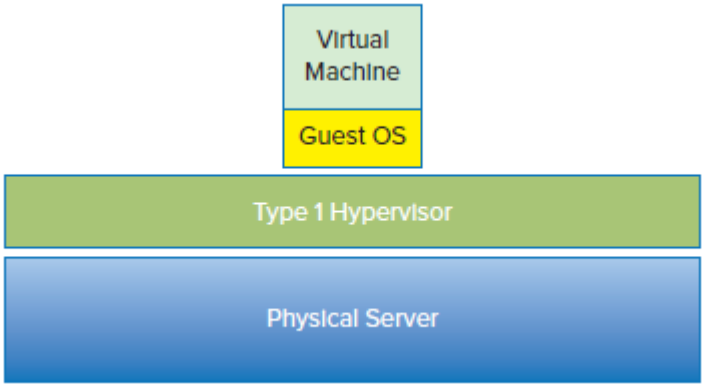
1.1.7.1. Tipos de hipervisor

Existen dos tipos de hipervisor o monitor de máquina virtual, el tipo 1 y 2.

1.1.7.1.1. Hipervisor tipo 1

La descripción dada por Portnoy: “El hipervisor tipo 1 corre directamente en el hardware del servidor sin un sistema operativo debajo de él”[8]. En la siguiente figura se ilustra la arquitectura de un hipervisor tipo 1.

Figura 2. **Arquitectura de hipervisor tipo 1**



Fuente: PORTNOY, Matthew. *Virtualization Essentials*. p. 22.

[8] PORTNOY, Matthew. *Virtualization Essentials*. p. 21.

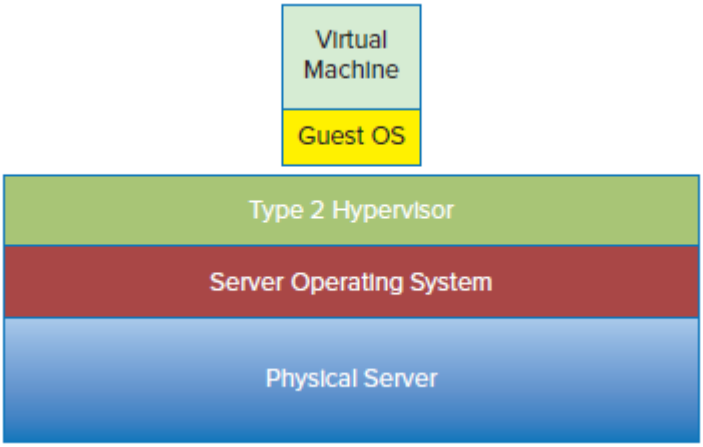
Ya que no hay una capa interviniendo entre el hipervisor y el hardware físico, también se conoce como hipervisor *bare-metal*.

Al hipervisor tipo 1 tiene mejor rendimiento y es considerado más seguro que el hipervisor tipo 2, ya que un *guest* no puede afectar al hipervisor ni a otros *guests*, lo que se traduce en que una máquina virtual solo puede dañarse a sí misma.

1.1.7.1.2. Hipervisor tipo 2

Portnoy proporciona la siguiente descripción: “Un hipervisor tipo 2 corre encima de un sistema operativo tradicional”[9]. La siguiente figura muestra la arquitectura de un hipervisor tipo 2.

Figura 3. **Arquitectura de hipervisor tipo 2**



Fuente: PORTNOY, Matthew. *Virtualization Essentials*. p. 23.

[9] PORTNOY, Matthew. *Virtualization Essentials*. p. 23.

Un beneficio de este hipervisor es que puede soportar una gran cantidad de hardware, porque es heredado del sistema operativo que usa. También un hipervisor de este tipo, es fácil de instalar y desplegar porque la mayoría de configuración de hardware funciona, como por ejemplo, la red y el almacenamiento.

Este hipervisor no es tan eficiente como el tipo 1, debido a la capa extra entre el hipervisor y el hardware; también es menos confiable, ya que cualquier cosa que afecte la disponibilidad del sistema operativo fundamental, también puede impactar en el hipervisor y los *guests* que soporta.

1.2. Tipos de virtualización

Tal como lo dice Bernard Golden, “La virtualización tiene un número de usos comunes, todo centrado alrededor del concepto que la tecnología representa una abstracción de los recursos físicos”[10]. Estos usos comunes son virtualización de servidores, de almacenamiento, de red y de aplicaciones.

1.2.1. Virtualización de servidores

David Williams en su libro *Virtualization with Xen* se menciona: “La virtualización de servidores es la forma dominante de virtualización más usada actualmente”[11]. Existen cuatro tipos de virtualización de servidores: completa, paravirtualización, virtualización nativa y virtualización por sistema operativo.

[10] GOLDEN, Bernard. *Virtualization for Dummies*. p. 25.
[11] WILLIAMS, et. al. *Virtualization with Xen*. p. 26.

1.2.1.1. Virtualización completa

Este tipo de virtualización permite correr en un ambiente especial (una máquina virtual) un sistema operativo sin modificar en la parte superior del sistema operativo *host*.

Williams define virtualización completa o *full virtualization* de la siguiente manera: “es una técnica de virtualización que provee una completa simulación de la capa de hardware”[12]. Esto indica que todo software que se ejecute en el hardware de la máquina *host* puede ejecutarse en la máquina virtual; cuenta con la más amplia variedad de sistemas operativos invitados o *guest*.

Algunos ejemplos de soluciones de virtualización completa son *Virtual Box*, *Quemu*, *Hiper-V*, etc.

1.2.1.2. Paravirtualización

El siguiente tipo de virtualización de servidores fue introducida por el *Xen Project Team*, el cual fue adoptado por otras soluciones de virtualización posteriormente. Williams establece la siguiente definición para este tipo de virtualización: “La paravirtualización es una técnica de virtualización que provee una simulación parcial de la capa de hardware”[13].

La mayoría, pero no todas las características de hardware son simuladas en este tipo de virtualización, y por lo tanto no todo el software que se ejecuta en la máquina *host* se puede ejecutar en la máquina virtual.

[12] GOLDEN, Bernard. *Virtualization for Dummies*. p. 27.
[13] WILLIAMS, et. al. *Virtualization with Xen*. p. 27.

1.2.1.3. Virtualización nativa o híbrida

Esta técnica es una combinación de virtualización completa y paravirtualización combinada con técnicas de aceleración de entrada y salida.

1.2.1.4. Virtualización por sistema operativo

Wikipedia proporciona la siguiente definición: “La virtualización por sistema operativo es un método de virtualización de servidores en el cual el *kernel* de un sistema operativo permite la existencia de múltiples instancias aisladas de espacio de usuario, en lugar de solo una”[14].

Este tipo de virtualización está basada en el concepto de contenedor, que se verá más a fondo en el capítulo 2.

1.2.2. Virtualización de almacenamiento

Según Bernard Golden la virtualización de almacenamiento “es un proceso de abstracción lógica del almacenamiento físico”[15].

Este tipo de virtualización puede ser implementado dentro de arreglos de almacenamiento propios o a nivel de red, donde múltiples discos dispersos a través de la red, pueden ser agrupados en un simple dispositivo de almacenamiento.

Algunas ventajas de este tipo de virtualización son migrar máquinas virtuales, alta disponibilidad, tolerancia a fallos y recuperación de desastres.

[14] Wikipedia.org. *Operating Level Virtualization*. Consulta: 18 de octubre de 2016
[15] GOLDEN, Bernard. *Virtualization for Dummies*. p. 16 – 17.

1.2.2.1. Tipos de virtualización de almacenamiento

Hay dos grandes tipos de sistemas de almacenamiento en red compartido donde se encontrará la virtualización de almacenamiento incorporada.

1.2.2.1.1. Network Attached Storage

Conforme Golden, un *Network Attached Storage* (*NAS*) es "un dispositivo de almacenamiento que se establece en la red y ofrece almacenamiento a los servidores en la red"[16]. Un *NAS* permite múltiples clientes, por ejemplo, usuarios y servidores para compartir archivos en la red local.

NAS utiliza protocolos basados en ficheros, como *NFS* o *SMB/CIFS*, en los cuales el almacenamiento es remoto y se realizan peticiones de archivos en lugar de un bloque de disco.

1.2.2.1.2. Storage Area Networks

Golden se refiere a un *Storage Area Network* (*SAN*) como "un dispositivo de almacenamiento accesible a servidores de tal manera que el dispositivo aparece localmente agregado al sistema operativo"[17]. Un *SAN* típicamente tiene su propia red de dispositivos de almacenamiento, utilizan conectividad por un canal de fibra, e *iSCSI*, que es un protocolo basado en *IP* para enlazar dispositivos de almacenamiento.

Una *SAN* no provee un nivel de abstracción a nivel de archivo como lo hace un *NAS*, solo permite operaciones a nivel de bloque.

[16] GOLDEN, Bernard. *Virtualization for Dummies*. p. 17.
[17] Ibid. p. 18.

1.2.3. Otros tipos de virtualización

Otro tipo de virtualización es una *VPN* (*Virtual Private Network*). *Cisco System* define una *VPN* como “una red privada construida dentro de una infraestructura de red pública”[18]. Generalmente una empresa utiliza una *VPN* para conectar de una manera segura sus oficinas y usuarios remotamente.

Virtualización de direcciones *IP* (*VIP*) constituye otro tipo de virtualización. Según *IBM* una *VIP* es “una manera de asignar una o varias direcciones al sistema sin la necesidad de enlazar la dirección con una interfaz física”[19].

1.2.4. Beneficios de la virtualización

El utilizar virtualización supone varios beneficios para un negocio. La empresa *VMware*[20] en su artículo *Virtualization: Overview*, proporciona una lista de dichos beneficios, que se listan a continuación:

- Reduce los costos operativos y de capital.
- Minimiza o elimina el tiempo de inactividad.
- Incrementa la productividad, eficiencia, agilidad y capacidad de respuesta de las tecnologías de la información.
- Provee aplicaciones y recursos más rápidamente.
- Soporta continuidad del negocio y recuperación de desastres.
- Simplifica la gestión del centro de datos.
- Construye un verdadero centro de datos definido por software.
- Las operaciones se automatizan.

[18] Cisco Systems. *VPN*. Consulta: 18 de octubre de 2016.
[19] IBM. *Direccionamiento con IP Virtual.* Consulta: 30 de octubre de 2016.
[20] VMware, Inc. *Virtualization: Overview*. Consulta: 30 de agosto de 2016.

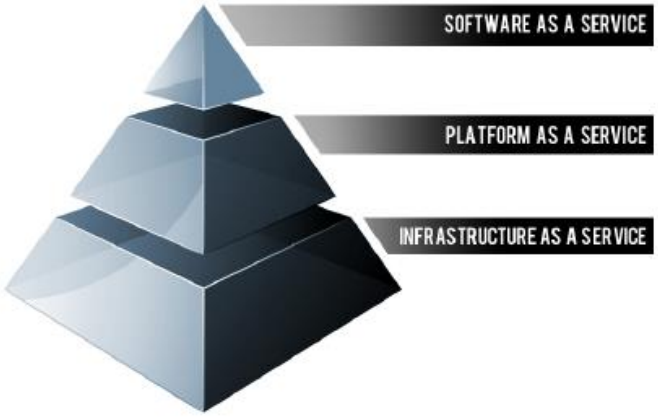
1.3. Computación en la nube

Hay muchas definiciones de lo que es computación en la nube o *cloud computing*. Una de las definiciones que mejor abarca todas las características de este concepto, es la que proporciona *Amazon Web Services* en su sitio oficial: "*Cloud computing* se refiere a la entrega bajo demanda de recursos informáticos y aplicaciones a través de Internet, con un sistema de precios basado en el consumo realizado"[21].

1.3.1. Modelos de computación en la nube

Existen tres modelos principales de computación en la nube. Cada modelo representa una parte diferente del *cloud computing stack*, como se puede observar en la siguiente figura.

Figura 4. ***Cloud computing stack***



Fuente: KEPES, Ben. *Understanding the Cloud Computing Stack*. Consulta: 6 de septiembre de 2016.

[21] Amazon. *What is Cloud Computing?.* Consulta: 18 de octubre de 2016.

1.3.1.1. Sofware as a Service (SaaS)

Según *Amazon Web Services*, este tipo de *cloud* “proporciona un producto completo que el proveedor del servicio ejecuta y administra.”[22]. Un ejemplo es un servicio de correo, en el cual el usuario no tiene que administrar la agregación de nuevas características ni mantener los servidores y los sistemas operativos en que está corriendo el servicio. El usuario acá, solo tiene que preocuparse en cómo debe utilizar el software concreto.

Este tipo de software utiliza un tipo de licenciamiento tanto como un servicio a pedido, a través de una suscripción o sin cargo, a menos que haya la oportunidad de generar ganancias.

1.3.1.2. Platform as a Service (PaaS)

Amazon Web Services propociona la siguiente descripción: “Las Plataformas como servicio eliminan la necesidad de las compañías de administrar la infraestructura subyacente (normalmente hardware y sistemas operativos) y le permiten centrarse en la implementación y la administración de sus aplicaciones”[23].

Una *Paas* es análogo a un *SaaS*, con la diferencia que este sirve para crear el software, en lugar de entregarlo ya hecho.

[22] Amazon. *Types of Cloud Computing.* Consulta: 19 de octubre de 2016.
[23] Ibid.

1.3.1.3. Infrastructure as a Service (IaaS)

Amazon Web Services, describe este tipo de *cloud* de la siguiente manera "La Infraestructura como servicio, que a veces se abrevia a *IaaS*, contiene los bloques de creación fundamentales para la TI en la nube"[24].

De manera general, un *IaaS* proporciona acceso a las propiedades de redes, a los servidores y al almacenamiento de datos.

1.3.2. Modelos de implementación de computación en la nube

Estos modelos representan la categoría del ambiente de la nube, su propósito y naturaleza. Son principalmente distinguidas por posesión, tamaño y acceso. Los modelos de implementación más comunes se describen a continuación.

1.3.2.1. Nube pública o externa

En el libro *Grid y Cloud Computing* se define de la siguiente manera: "Una nube pública es un centro de datos dirigido por terceros"[25]. Este tipo de nubes pueden ir dirigidas tanto para empresas como para el público en general.

Su acceso es público, es decir que cualquier persona con conexión a internet puede acceder a ellos.

[24] Amazon. *Types of Cloud Computing.* Consulta: 19 de octubre de 2016.
[25] STANOEVSKA-SLAVENA, et. al. *Grid and Cloud Computing - A Business Perspective on Technology and Applications*. p. 57.

1.3.2.2. Nube privada o interna

Una nube privada le pertenece a una sola compañía, y esta tiene control total de las aplicaciones que corre la infraestructura de dicha nube y solo puede accederse internamente dentro de la compañía.

Según Stanoevska-Slavena, Wozniak y Ristol, autores del libro *Grid and Cloud Computing*, la razón por la que las empresas utilizan una nube privada es la siguiente "Una compañía no está dispuesta a soportar los riesgos asociados de moverse a una nube pública y por lo tanto construyen nubes internas a manera de beneficiarse del *cloud computing*"[26].

1.3.2.3. Nube híbrida

Según los autores del libro *Grid and Cloud Computing*, una nube híbrida es aquella que "combina nubes privadas y públicas, permitiendo que ambas corran algunas aplicaciones en una infraestructura de nube privada y otras en una nube pública"[27].

El utilizar una nube híbrida, agrega algunos retos que tiene que ser considerados durante la fase de planeación y diseño. Algunos de estos retos son la distribución de aplicaciones, la seguridad, etc.

1.3.3. Beneficios

No importando que proveedor se utilice los beneficios del *cloud computing* son en la mayoría similares. De manera general, dichos beneficios los

[26] STANOEVSKA-SLAVENA, et. al. *Grid and Cloud Computing - A Business Perspective on Technology and Applications*. p. 57.

[27] Op. Cit. p. 58.

presentan muchos proveedores de servicios *cloud*. A continuación, se listan los enunciados por *Amazon Web Services*[28]:

- Facilidad de uso
- Flexible
- Rentable
- De confianza
- Escalabilidad y alto desempeño
- Seguro

1.3.4. Proveedores

Un proveedor de *cloud computing* es una compañía que ofrece un componente, ya sea de *SaaS*, *PaaS* o *IaaS* a otras compañías o personas individuales. A veces es referido como *Cloud Service Provider (CSP)*. A continuación, se listan algunos proveedores.

Tabla I. **Listado de proveedores de *cloud computing***

SaaS	***PaaS***	***IaaS***
Oracle On Demand	*Engine Yard*	*CloudSigma*
SalesForce	*Windows Azure*	*Linode*
NetSuite	*Lunacloud*	*Amazon Web Services*
AppDynamics	*Qt Cloud Services*	*Dimension Data*
Aprenda	*Skyvia*	*Rackspace*
Cloud9 Analytics	*CloudForge*	*Digital Ocean*
Cumulux	*Openstack*	*IBM Cloud*
Intacct	*Amazon Elastic Beanstack*	*CloudForge*
Etc.	Etc.	Etc.

Fuente: elaboración propia.

[28] Amazon. *Benefits at a Glance*. Consulta: 21 de octubre de 2016.

1.4. Patrones y estilos de arquitectura

A continuación se describe qué es un patrón de diseño, qué es un estilo de arquitectura y cuáles son los principales estilos que entran en debate para esta investigación.

1.4.1. Patrón de diseño

Un patrón de diseño se podría decir que es una plantilla, en la cual se listan los pasos de cómo se tiene que resolver un problema y que puede ser usado en muchas situaciones diferentes. En el libro *A Pattern Language* presenta la siguiente definición: “Un patrón de diseño describe un problema el cual ocurre una y otra vez en cualquier ambiente, y luego describe la esencia de la solución a ese problema, de tal manera que esa solución se pueda usar un millón de veces más, sin hacerlo de la misma manera dos veces”[29].

1.4.2. Arquitectura de software

Según la *IEEE* una arquitectura de software es “la organización fundamental de un sistema incorporado en sus componentes, sus relaciones con los demás, y con el ambiente, y los principios que guían su diseño y evolución”[30].

Una arquitectura de software que este bien realizada facilita la compresión a toda persona que tenga contacto con el sistema de gran manera, por lo que se tiene que tomar en cuenta el entorno interno del sistema como el entorno externo.

[29] ALEXANDER, et. al. *A Pattern Language*. p. 10.
[30] Institute of Electrical and Electronics Engineers, Inc. *IEEE Std 1471-2000*. p. 3.

1.4.3. Arquitectura monolítica

Una arquitectura monolítica es construida como una única unidad, por ejemplo, comúnmente una aplicación empresarial se compone de tres partes: un modelo que representa el acceso a los datos, una vista que representa una interfaz de usuario y un controlador que comunica la interfaz de usuario con el modelo de datos; estas tres partes se dice que forman un sistema monolítico, puesto que representan un solo archivo ejecutable.

Por definición, en una arquitectura monolítica solamente se emplea una tecnología de desarrollo, lo cual limita la disponibilidad de una herramienta adecuada para cada tarea que debe ejecutar el sistema.

1.4.4. Arquitectura de microservicios

James Lewin y Marting Fowler, ambos expertos en el campo, definen una arquitectura de microservicios de la siguiente manera: “El estilo de arquitectura de microservicios es un enfoque para desarrollar una aplicación individual como un conjunto de pequeños servicios, cada uno corriendo su propio proceso y comunicándose con mecanismos livianos”[31]. A menudo la comunicación entre las aplicaciones se realiza por medio de un recurso *HTTP*.

En el capítulo 3 se profundiza en los conceptos de la arquitectura de microservicios.

[31] LEWIS, et. al., 2014. *Microservices.* Consulta: 7 de septiembre de 2016.

2. *DEVOPS* Y CONTENEDORES

En este capítulo se explican los conceptos sobre los contenedores, además de conceptos relacionados con metodologías de desarrollo de software que se acoplan con el uso de contenedores para obtener mejores resultados y lograr con mayor facilidad sus objetivos.

2.1. *Development and Operations (DevOps)*

En el libro *DevOps for Dummies* escrito por Sanjeev Sharma se define *DevOps* de la siguiente manera "*Development and Operations (DevOps)* es un enfoque basado en principios de agilidad y eficiencia en los que los propietarios de empresas y los departamentos de desarrollo, operaciones y control de calidad colaboran para ofrecer software de forma continua, lo que permite a las empresas sacar partido de las oportunidades del mercado de forma más rápida y reducir el tiempo para incluir respuestas de clientes"[32].

Tradicionalmente las áreas de las tecnologías de la información se han organizado en dos áreas con una obligación concreta. Un área denominada desarrollo a cargo de crear software. La otra se denomina operación (o sistemas), que se encarga de proveer la infraestructura para que el software esté disponible y sea accesible a los usuarios. En un sentido amplio, el objetivo principal de *DevOps* es que los objetivos de desarrollo y los de operación se alineen para que en conjunto coincidan con los del negocio.

[32] SHARMA, Sanjeev. *DevOps para Dummies*. p. 1.

DevOps no constituye una metodología como lo es *Scrum* o *XP*, *DevOps* se enfoca en una manera de pensar, una cultura, que cada negocio debe adoptar según sus necesidades.

2.1.1. Rutas de adopción a *DevOps*

La arquitectura de *DevOps*, propone cuatro rutas de adopción, descritas a continuación.

2.1.1.1. Planificación y medición

Esta ruta de adopción se centra en la planificación empresarial continua. El enfoque principal es establecer los objetivos del negocio y ajustarlos en base a la retroalimentación de los clientes.

Para alcanzar esto, “las empresas deben ser ágiles y ser capaces de reaccionar de forma rápida antes los comentarios de los clientes”[33]. Se requiere planificar y re-planificar rápidamente, mientras se maximiza la habilidad de crear valor.

2.1.1.2. Desarrollo y pruebas

Esta ruta de adopción forma el núcleo de las funciones de desarrollo colaborativo y control de calidad.

El desarrollo colaborativo permite a los profesionales que son parte de equipos multifuncionales trabajar en forma conjunta, ofreciendo un conjunto habitual de prácticas y una plataforma común para crear y entregar software.

[33] SHARMA, Sanjeev. *DevOps para Dummies*. p. 10.

Una función primordial a incluir en todo desarrollo colaborativo es la integración continua, porque “añade un gran valor a *DevOps*, ya que permite que grandes equipos de desarrolladores trabajen en componentes de varias tecnologías en diversas ubicaciones”[34].

Por otro lado, se tienen las pruebas continuas, que no son más que “pruebas anticipadas y constantes durante todo el ciclo de vida. Esto se consigue empleando funciones como virtualización de servicios y pruebas automáticas”[35].

2.1.1.3. Lanzamiento de versiones y despliegues

En esta ruta se forman la mayoría de funciones esenciales de *DevOps*, llevando la integración continua a otro nivel.

Acá se forma crear un canal de distribución que “facilita el despliegue continuo de software a control de calidad y después a producción de forma eficiente y automatizada. El objetivo del lanzamiento de versiones y de la implementación continua es presentar funciones a los clientes y usuarios lo antes posible”[36].

2.1.1.4. Supervisión y optimización

Esta ruta comprende dos prácticas, supervisión continua y optimización continua, que ayudan a las empresas a monitorizar los resultados de aplicaciones desplegadas en el ambiente de producción y la retroalimentación de los clientes.

[34] SHARMA, Sanjeev. *DevOps para Dummies*. p. 21.
[35] Op. Cit. p. 12.
[36] Ibíd.

El estar supervisando constantemente menciona Sharma que “ofrece datos y métricas al personal de líneas de negocios, desarrollo, control de calidad y operaciones, así como de otras partes interesadas, sobre aplicaciones en distintas fases del ciclo de entrega”[37].

Un equipo de entrega software, recibe dos tipos importantes de información. El primer tipo de información relevante es como los clientes utilizan el software. El segundo de información es la retroalimentación que un cliente hace tras utilizar el software. Este ciclo de información continua, permite a las organizaciones estar optimizando continuamente.

2.1.2. Integración continua

Sharma se refiere a la integración continua “un ejercicio en el que los desarrolladores integran de forma frecuente su trabajo con el de otros miembros del equipo de desarrollo”[38]. Ya que se valida el trabajo integrado de varios equipos, se reduce el riesgo y se previenen problemas durante el ciclo de vida de desarrollo del software.

2.1.3. Entrega continua

De forma inherente la integración continua implica la práctica de la entrega continua. Según Sharma “el proceso de automatizar el despliegue del software a entornos de pruebas, pruebas de sistemas, preproducción y producción”[39].

[37] SHARMA, Sanjeev. *DevOps para Dummies*. p. 13.
[38] Op. Cit. p. 11.
[39] Op. Cit. p. 21.

2.1.4. Despliegue continuo

La entrega continua, no significa que cada cambio que pase satisfactoriamente las validaciones, sea desplegado a producción. Según Carl Caum describe al despliegue continuo como el siguiente paso de la entrega continua ya que “cada cambio que pasa las pruebas automatizadas se despliega a producción automáticamente”[40].

El despliegue continuo puede que no se acople a muchas empresas por leyes o por otros requerimientos, ahora bien, la entrega continua es un requisito imperante para las prácticas de *DevOps*.

2.2. Contenedores

En el contexto de software, el término contenedor es “una referencia generalizada para cualquier partición virtual que no sea una máquina virtual basada en un hipervisor”[41].

Los contenedores permiten empaquetar y aislar aplicaciones con todas las librerías y dependencias que necesitan para poder ejecutarse.

2.2.1. Técnicas de partición de recursos no basadas en hipervisores

Intel en su artículo *Linux Containers Streamline Virtualization and Complement Hypervisor-Based Virtual Machines*, comenta que “los enfoques alternativos para la partición de recursos que no se basen en hipervisores,

[40] CAUM, Carl. *Continuous Delivery Vs. Continuous Deployment: What's the Diff?*. Consulta: 22 de octubre de 2016.

[41] Intel. *Clear Containers*. Consulta: 22 de octubre de 2016.

fueron desarrollados por *Unix* y a medida que los sistemas operativos tipo *Unix* crecieron a finales de 1970 y principios de 1980 también fueron impulsados por diferentes organizaciones que incluían laboratorios *Bell*, *AT&T*, Digital Equipment Corporation, Sun Microsystems e instituciones académicas. Estas tecnologías desarrolladas se refirieron colectivamente como virtualización por sistema operativo, proporcionando enfoques livianos de virtualización, actuando como los fundamentos de los contenedores *Linux* de hoy en día"[42].

2.2.1.1. Llamada al sistema *chroot*

Este es un fundamento crítico en la virtualización basada en contenedores. Tal como lo define *Intel* "el mecanismo *chroot* puede redefinir el directorio raíz para cualquier programa en ejecución, previniendo con eficacia que el programa sea capaz de nombrar o acceder recursos fuera del árbol del directorio raíz"[43].

Esta característica fue introducida en la versión 7 de *Unix* por los laboratorios *Bell* en 1979 y como parte de 4.2*BSD* por la Universidad de California, *Berkeley* en 1983.

2.2.1.2. FreeBSD jails

FreeBSD jails es parecido en concepto que *chroot*, pero con un enfoque en la seguridad. Según *Intel* "las definiciones de *FreeBSD jails* pueden restringir explícitamente el acceso fuera del entorno de espacio aislado por entidades tales como archivos, procesos, y cuentas de usuario"[44].

[42] Intel. *Clear Containers*. Consulta: 22 de octubre de 2016.
[43] Ibid.
[44] Ibid.

2.2.1.3. Solaris Containers (Solaris Zones)

Este mecanismo se hizo basado en las capacidades de *chroot* y *FreeBSD jails*. Según *Intel* las *Zones*, “son instancias de servidores individuales que pueden coexistir dentro de una sola instancia de sistema operativo”[45].

Sun Microsystems introdujeron esta característica con el nombre de *Solaris Containers* como parte de *Solaris* 10 en el año 2005, luego *Oracle* oficialmente cambio el nombre a *Solaris Zones* con el lanzamiento de *Solaris* 11 en el año 2011.

2.2.1.4. Contenedores *Linux* (*LXC*)

Los contenedores *Linux* es un proyecto de código abierto que se introdujo en el año 2007. Estos ofrecen virtualización basada en sistema operativo.

Oracle en su artículo *Consolidate with Oracle Linux Containers* proporciona la siguiente descripción: “Los contenedores *Linux* (*LXC*) permiten correr múltiples instancias de *Linux* aislados (contenedores) en el mismo *host*. Un contenedor es una manera de aislar un grupo de procesos de otros en un sistema *Linux* en ejecución. Al hacer uso de funcionalidades existentes como la nueva gestión de recursos del *kernel* de *Linux* y las características de aislamiento de recursos (*cgroups* y *namespaces*), estos procesos pueden tener su propia vista privada del sistema operativo con su propio *ID* de proceso (*PID*), sistema de archivos e interfaces de red”[46].

[45] Intel. *Clear Containers*. Consulta: 22 de octubre de 2016.
[46] Oracle. *Linux Containers.* Consulta: 22 de octubre de 2016.

2.2.1.4.1. Cgroups

Kernel control groups, mayormente conocido solo como *control groups*, son una característica del *kernel* de *Linux*, que “permiten distribuir recursos (tales como tiempo de *CPU*, memoria del sistema, ancho de banda, o una combinación de estos recursos) entre grupos definidos por el usuario de tareas (procesos) corriendo en un sistema”[47].

Cada grupo de recursos puede ser configurado manualmente o de manera programada, con lo cual se gana control sobre la distribución, priorización, denegación y monitoreo de los recursos del sistema.

2.2.1.4.2. Namespaces

SUSE en su artículo *Virtualization with Linux Containers (LXC)* describe a los *namespaces* como “una característica del *kernel* para aislar algunos recursos como red, usuarios, y otros para un grupo de procesos”[48].

Con esto se logra que contenedores individuales puedan tener su propia interfaz y dirección *IP*. También ayudan a limitar la vista que tiene cada *cgroup* de los recursos del sistema.

2.3. *Docker*

James Turnbull en su libro *The Docker Book*, describe a *Docker* como “un motor de código abierto que automatiza el despliegue de aplicaciones en

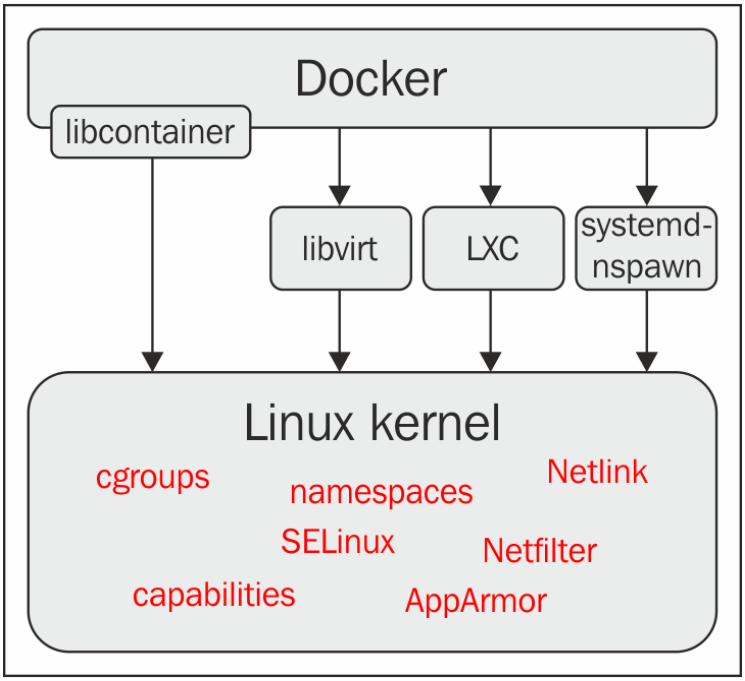
[47] Red Hat Inc. *Chapter 1. Introduction to Control Groups (Cgroups)*. Consulta: 22 de octubre de 2016.

[48] SUSE. *LXC Quickstar*. Consulta: 14 de julio de 2016.

contenedores”[49]. *Docker* fue desarrollado por la empresa del mismo nombre *Docker, Inc.*, anteriormente conocida como *dotCloud Inc*.

Docker utiliza una capa de abstracción para poder acceder a las capacidades de virtualización (*cgroups*, *namespaces*, *capabilities*, etc.) del *kernel* de *Linux*, tal como se muestra en la siguiente figura.

Figura 5. **Ambientes de ejecución de *Docker***



Fuente: VADUVA, Alexandru. *Virtualization*. Consulta: 24 de octubre de 2016.

El ambiente de ejecución por defecto que utilizaba *Docker* antes de la versión 0,9, era *LXC*, pero la implementación de este cambia mucho en varias distribuciones, lo cual representaba un problema, ya que lo que *Docker Inc.*

[49] TURNBULL, Jame. *The Docker Book*. p. 8.

buscaba era una manera de estandarizar la manera en que se crean contenedores dentro de un sistema operativo. Desde la versión 0,9, *Docker* incluye su propio ambiente de ejecución, llamado *libcontainer*. Todos los esfuerzos de este nuevo ambiente, van enfocados en convertirlo en un estándar, además de volver a *Docker* una tecnología multiplataforma.

2.3.1. Historia de *Docker*

El ingeniero Solomon Hykes, cofundo junto con el ingeniero Sébastien Pahl *dotCloud*. Su "software ofrecía una plataforma para poder codificar en la nube de *Amazon*" [50]. La empresa recaudó $11 millones de varios inversores, por ejemplo, Jerry Yang cofundador de *Yahoo*, y firmas como *Trinity Ventures* y *Benchmark*. Después de esto, la empresa se vino abajo, lo clientes bajaron, y el soporte de *Amazon* mejoro, con lo cual el crecimiento de la empresa comenzó a disminuir.

La empresa comenzó a mejorar, cuando Ben Golub (en ese entonces presidente de la empresa *Plaxo*) se unió a la compañía. Golub apoyo el proyecto de Hykes de enfocarse en la gestión de contenedores, la cual usaban como tecnología fundamental en el software de *dotCloud*. Golub y Hykes realizaron un movimiento aún más atrevido: dar de manera gratis su tecnología fundamental para que pudiera ser usada por cualquier persona. En los siguientes meses, cientos de voluntarios estaban escribiendo código para extender y mejorar la manera en que se ejecutan los contenedores.

Fue en la primavera del año 2013, en una conferencia hecha en Santa Clara, California, donde se anunció el proyecto *Docker*. Para finales del mismo

[50] KONRAD, Alex. *How Docker Escaped Near-Death to Become Software's Next Big Thing*. Consulta: 24 de octubre de 2016.

año, su popularidad impresionó tanto a nuevos inversores, logrando obtener $15 millones. Desde entonces, *Docker* se ha venido expandiendo, ya cuenta con versiones para *Linux*, *Windows* y *Mac*, además de servicios empresariales.

2.3.2. Componentes base

A continuación, se describen los componentes núcleo de *Docker*, los cuales son el servidor y el cliente, lo cuales en conjunto se les conoce como *Docker Engine*.

2.3.2.1. Servidor Docker

También se le conoce como *Docker daemon*. La documentación oficial lo define como "el proceso persistente que crea y gestiona los objetos *Docker*, tales como imágenes, contenedores, redes y volúmenes de datos"[51].

El servidor *Docker* está integrado en el mismo binario que se usa para ejecutar los procesos del cliente, por ello tiende a confundirse. El *Docker daemon* sin embargo necesita privilegios *root* para poder ejecutarse, mientras que el cliente no.

2.3.2.2. Cliente Docker

Gath Schulte, quien es creador de varios video cursos en *CBT Nuggets*, define el cliente *Docker* como una "interfaz de línea de comandos para interactuar con el *Docker daemon*"[52].

[51] Docker Inc. *Daemon.* Consulta: 24 de octubre de 2016
[52] SHULTE, Garth. *What is Docker?* Consulta: 24 de octubre de 2016.

Docker utiliza una interfaz de tipo *RESTful*, para poder comunicar el cliente y el servidor. El cliente es usado en la mayoría del flujo de trabajo de *Docker* y para comunicarse con otros servidores remotos.

2.3.3. Componentes de flujo de trabajo

Para entender el funcionamiento interno de *Docker* es necesario conocer sobre las imágenes, contenedores y registros *Docker*.

2.3.3.1. Imagen Docker

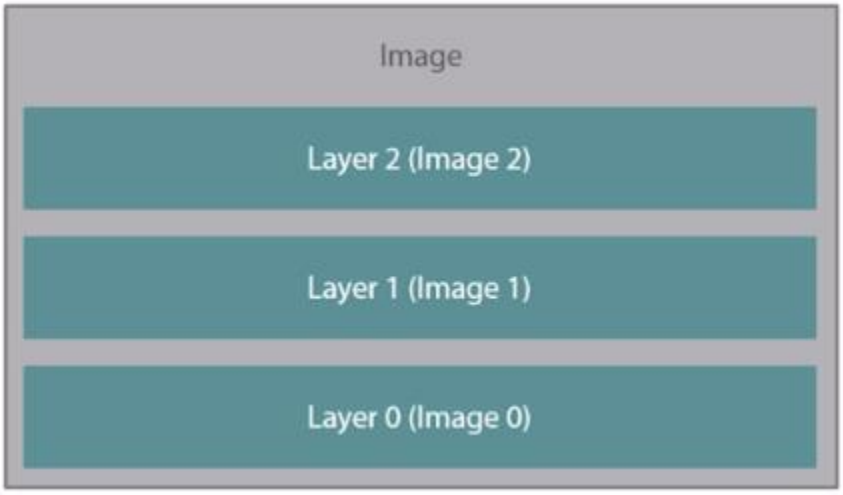
Son una parte fundamental de *Docker*, ya que sin una imagen no se puede crear un contenedor *Docker*. De la documentación oficial de se puede extraer el siguiente concepto “una imagen *Docker* es una plantilla de solo lectura con instrucciones para crear un contenedor *Docker*”[53].

Docker puede construir imágenes manualmente o automáticamente utilizando las instrucciones de un *Dockerfile*. Un *Dockerfile* es un documento de texto que contiene todos los comandos que un usuario puede llamar desde la interfaz de línea de comandos para ensamblar una imagen.

Una imagen está compuesta de una o múltiples capas (imágenes intermedias) y metadatos importantes que representan todos los archivos requeridos para ejecutar un contenedor *Docker*. Tal como se muestra en la figura siguiente, se tres imágenes intemedias, las cuales juntas forman una sola imagen.

[53] Docker Inc. *Docker Overview.* Consulta: 24 de octubre de 2016.

Figura 6. **Estructura de una imagen *Docker***



Fuente: POULTON, Nigel. *Docker Deep Dive.* Consulta: 26 de octubre de 2016.

2.3.3.2. Contenedor *Docker*

El sitio oficial de *Docker* define un contenedor *Docker* como “una instancia ejecutable de una imagen *Docker*”[54].

Un contenedor específico, solo puede existir una vez, pero se pueden crear múltiples contenedores de una misma imagen.

2.3.3.3. Registro *Docker*

Cuando se tiene una imagen finalizada, el próximo paso es buscar un lugar donde guardar esta imagen, para esto existen los registros. *Docker Inc*. Lo define como “Un registro *Docker* simplemente es “una librería de imagenes”[55].

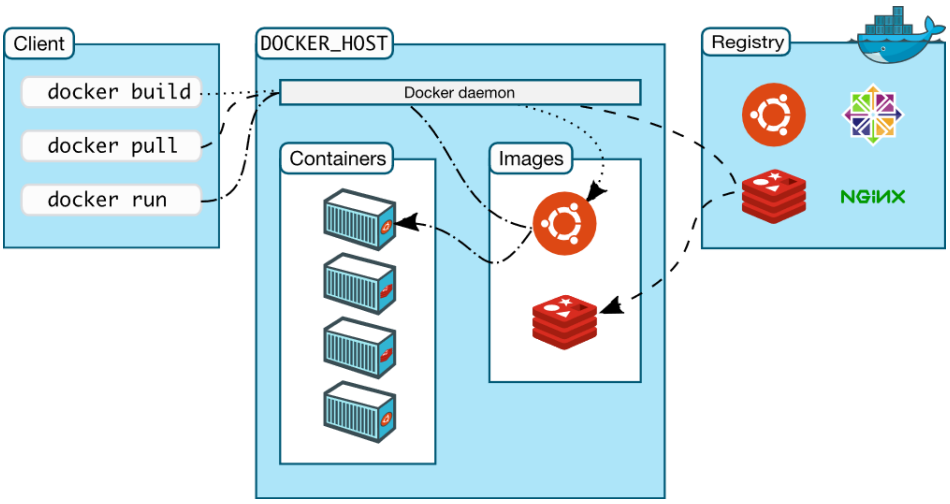
[54] Docker Inc. *Docker Overview.* Consulta: 24 de octubre de 2016.
[55] Ibid.

Un registro puede ser público o privado, y puede estar tanto en un servidor local como en uno remoto. *Docker Inc.* provee un repositorio llamado *Docker Hub*, para almacenar imágenes en la nube.

2.3.4. Arquitectura interna de *Docker*

Docker utiliza una arquitectura cliente-servidor. En la siguiente figura se muestra la interacción entre cada uno de los componentes su arquitectura.

Figura 7. **Arquitectura interna de *Docker***



Fuente: Docker Inc. *Docker Overview.* Consulta: 24 de octubre de 2016.

El cliente se comunica con el servidor, el cual es el encargado construir, ejecutar y distribuir los contenedores; para ello hace uso de imágenes, es de notar de la figura que la imagen de *Ubuntu* se tiene localmente, y la imagen de *Redis* se está descargando de un registro.

2.3.5. Beneficios

El segundo contribuyente más importante del proyecto *Docker*, *Red Hat*[56], proporciona una lista de los beneficios más relevantes que trae a las organizaciones el utilizar *Docker*. Dichos beneficios se listan a continuación:

- Despliegue rápido de aplicaciones
- Portabilidad a través de las máquinas
- Control de versiones y reúso de componentes
- Permite el trabajo compartido
- Un modelo ligero y un mínimo de gastos
- Mantenimiento simplificado

2.3.6. *Docker* una tendencia

Según una encuesta hecha por *Red Hat*[57], un 35% de sus clientes planean utilizar *Docker* como su plataforma para ayudar a los desarrolladores a entregar aplicaciones rápidamente.

La tendencia de *Docker*, se basa fundamentalmente en la facilidad en que permite mover una aplicación entre diferentes ambientes (desarrollo, pruebas, producción, etc.), mientras la funcionalidad se mantiene intacta.

Otro aspecto a tomar en cuenta del auge del uso de *Docker*, es el apoyo que está recibiendo de grandes compañías, como lo son *Amazon*, *IBM*, *Google*, e incluso *Microsoft*. Muchas de estas empresas ya han agregado soporte para *Docker* en sus plataformas, por ejemplo, *Amazon Web Services* y *Google*

[56] Red Hat Inc. *Understanding Linux containers*. Consulta: 25 de octubre de 2016.
[57] Ibid.

Cloud, con lo que las posibilidades para implementar *Docker* se expanden, haciendo una tecnología de mayor auge en las industrias.

2.4. *Docker* con *DevOps*

La piedra angular de *DevOps* es la integración continua, la cual comúnmente se trabaja con *Jenkins*. Esta herramienta ayuda al equipo de desarrollo a gestionar la construcción de las aplicaciones, ya que, en la mayoría de casos, se trabaja el desarrollo de manera distribuida.

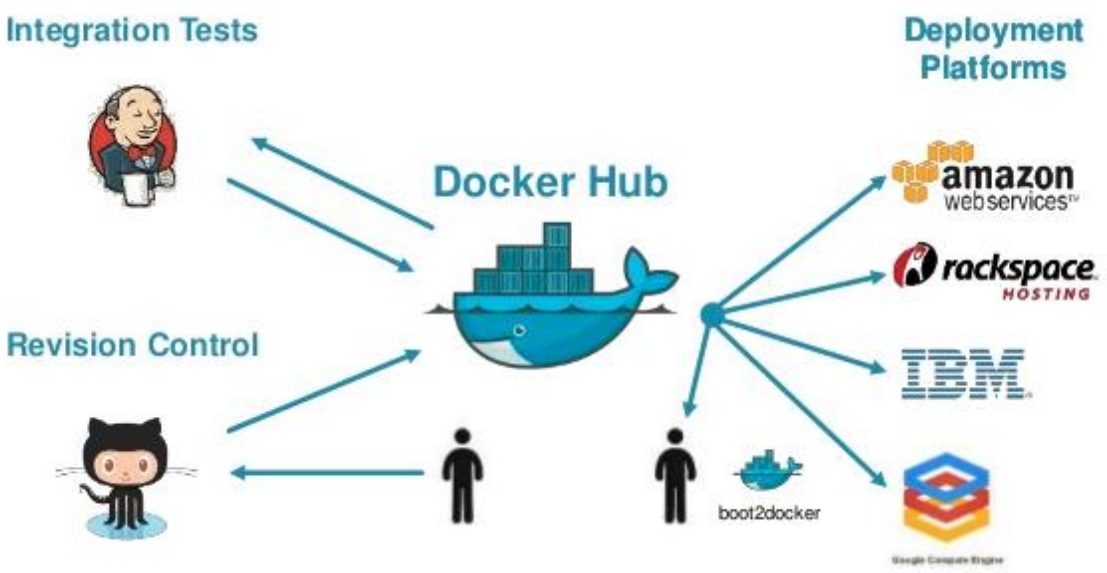
La construcción, el empaquetado y el despliegue de las aplicaciones, juegan un papel crucial en la cultura *DevOps*. Es acá, donde entra a la ecuación *Docker*, como una herramienta de automatización de construcción. En lugar de desplegar un conjunto de artefactos (por ejemplo, un archivo *EXE* o un archivo JAR) al ambiente objetivo, el equipo de operaciones ahora puede empaquetar la aplicación completa como una imagen *Docker*, la cual se sube a un registro en el cual puede ser accedida desde varios ambientes para el despliegue final.

El contribuyente de la revista *Forbes*, Janakiram MSV, en su artículo *How Docker Has Changed the DevOps Game*, indica que "La combinación de *Jenkins* con *Docker* es muy valorada para los equipos *DevOps*. Al aprovechar la estrecha integración con mecanismos de control de código fuente como *Git*, Jenkins puede iniciar un proceso de construcción cada vez que un desarrollador realice un *commit* a su código. Este proceso da como resultado una nueva imagen *Docker* que es disponible al instante alrededor de todos los ambientes. Las organizaciones están desplegando los registros privados de *Docker* para publicar y mantener sus imágenes internas"[58].

[58] MSV, Janakiram. *How Docker Has Changed the DevOps Game*. Consulta: 25 de octubre de 2016.

En la siguiente figura se presenta, un ejemplo de cómo *Jenkins* puede integrarse con *Docker* para automatizar el proceso de construcción y despliegue de aplicaciones.

Figura 8. **Integración de *Docker* y *DevOps***



Fuente: MSV, Janakiram. *How Docker Has Changed the DevOps Game*. Consulta: 25 de octubre de 2016.

3. ARQUITECTURA DE MICROSERVICIOS

En este capítulo se definen los conceptos relacionados a los microservicios, las ventajas y desventajas que conlleva usarlos, además se listan los casos en que conviene utilizar una arquitectura de microservicios. Por último, se presentan los pasos de implementación de este tipo de arquitectura.

3.1. Visión general

Una arquitectura de microservicios, tal como su nombre lo indica, tiene un alto énfasis en los pequeños servicios, como el componente primario de la arquitectura, usado para implementar y ejecutar las funcionalidades del negocio. Este tipo de arquitectura, ha emergido como un patrón común de desarrollo de software de diversas prácticas de grandes empresas, por ejemplo, *Amazon* y *Netflix*.

Según Martin Fowler y James Lewis en su artículo *Microservices a Definition of this New Architectural Term*, definen la arquitectura de microservicios como “un enfoque para el desarrollo de una sola aplicación como un conjunto de pequeños servicios, cada uno se ejecuta en su propio proceso y se comunican con mecanismos ligeros, a menudo por el protcolo *HTTP*”[59].

A los servicios mencionados en la definición anterior, se les conoce como microservicios. Del libro *Microservice Architecture - Aligning Principles, Practices and Culture* se extrae la siguiente definición “Un microservicio es un componente independiente de despliegue de alcance limitado que soporta la

[59] POSTA, Cristian. *Microservices for Java Developers*. p. 6.

interoperabilidad a través de la comunicación basada en mensajes"[60]. Los microservicios generalmente son implementados y operados por equipos pequeños con la suficiente autonomía que cada equipo y servicio puede cambiar sus detalles de implementación (incluso un reemplazo total) con un mínimo de impacto en el resto del sistema.

Las aplicaciones de microservicios tienen algunas características importantes en común. A continuación, se presenta el listado de estas características presentadas en el libro *Microservice Architecture*[61]:

- Pequeñas en tamaño
- Mensajería activada
- Limitado por contextos
- Desarrollo de manera autónoma
- Despliegue independiente
- Descentralizado
- La construcción y el *reléase* es hecho por procesos automáticos

Una arquitectura de microservicios tiene la característica esencial de ser distribuida, aspecto a tomar en cuenta, ya que se agrega complejidad a su implementación.

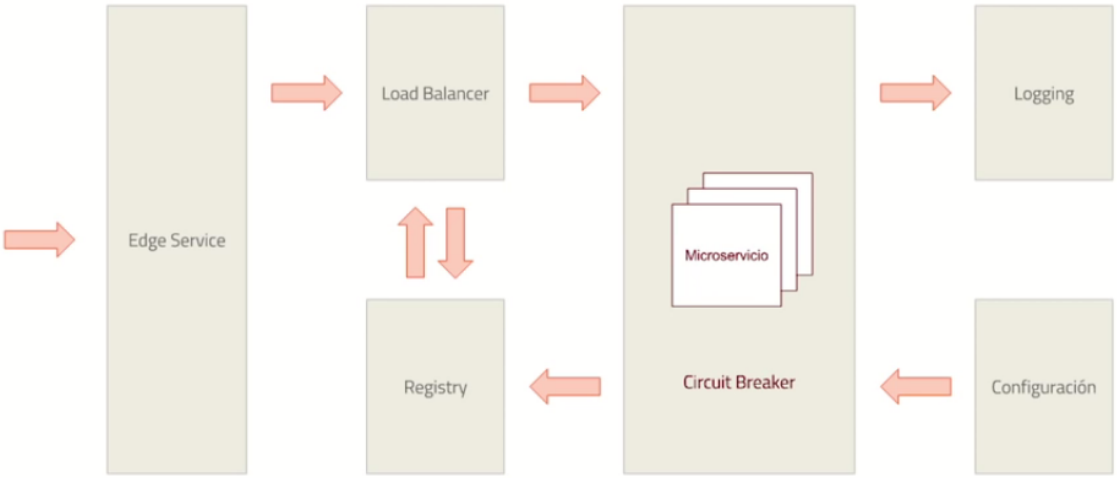
3.2. Componentes

En una aplicación de microservicios además de los servicios del negocio, involucra otros componentes que ayudan a que, en conjunto, formen una aplicación completa.

[60] NADAREISHVILI, et. al. *Microservice Architecture*. p. 6.
[61] Op. Cit. p. 7.

En la figura siguiente, se puede observar la interacción entre los microservicios del negocio y los componentes que los unen.

Figura 9. **Componentes básicos en una arquitectura de microservicios**



Fuente: GARRIDO, et. al. *Arquitecturas basadas en microservicios.* Consulta: octubre 2016.

3.2.1. *Edge service*

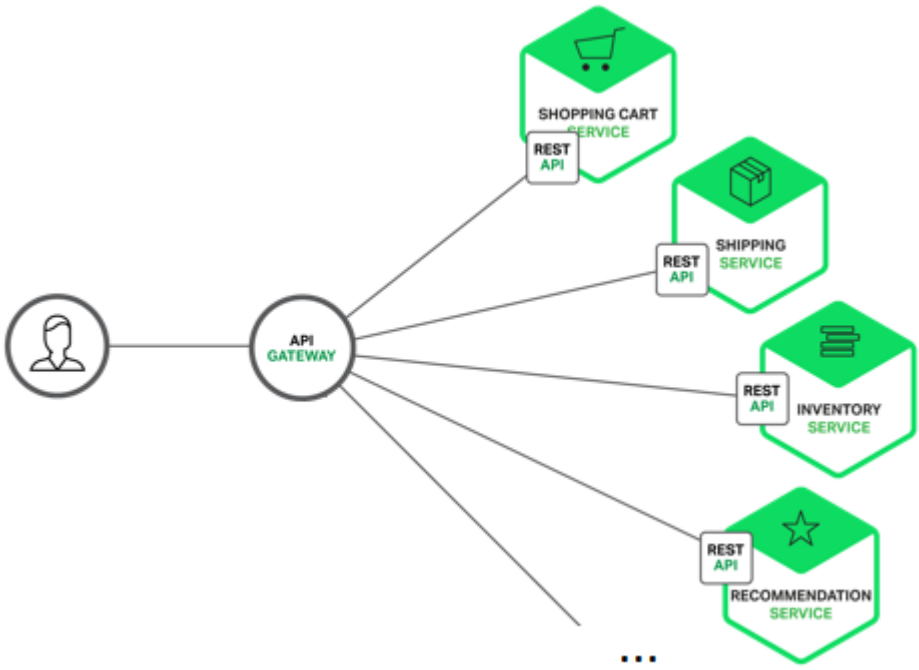
Un *edge service* también se conoce como servicio de enrutamiento. Este servicio permite a los clientes de una aplicación de microservicios, acceder a servicios individuales, por un único punto de entrada.

Richardson y Smith, proporcionan una definición más formal “un servicio de enrutamiento es una puerta de enlace para un único punto de entrada, gestiona todas las peticiones de los clientes, y realiza un re direccionamiento al servicio o servicios apropiados”[62].

[62] RICHARDSON, et. al. *Nginx*. p. 15.

En la figura siguiente se muestra cómo un servicio de enrutamiento re-direcciona las peticiones de un cliente hacia un determinado servicio.

Figura 10. **Uso de un servicio de enrutamiento**



Fuente: RICHARDSON, et. al. *Nginx*. p. 16.

Este servicio es importante para abstraer a todos los microservicios del tráfico externo; mapear las *URLs* de cada microservicio y permitir redirigir peticiones concretas a servicios concretos.

Algunas de las implementaciones de servicios de enrutamiento más utilizadas son: *API Gateway* (*AWS*), *HA-Proxy*, *Zuul* (parte de *Spring Cloud Netflix*).

3.2.2. Balanceador de carga y registro de servicios

Un cliente, para poder realizar una petición necesita conocer la ubicación de red (dirección *IP* y puerto) de la instancia del microservicio. Esto puede resultar un problema difícil de resolver, ya que las ubicaciones de red cambian y el conjunto de instancias de los microservicios cambian dinámicamente (por escalamiento, fallas o actualizaciones).

Precisamente, para resolver el problema anterior, se utiliza un servicio de descubrimiento (*discovery service*). Según Chris Richardson y Floyd Smith, autores del documento *Microservices – From Design to Deployment*, publicado por *Nginx,* un servicio de descubrimiento es “el responsable de determinar la ubicación de red de las instancias de los servicios disponibles”[63].

Este servicio de descubrimiento está formado por un balanceador de carga y un registro de servicios.

Un balanceador de carga, según la empresa *Citrix Systems, Inc.* es “un dispositivo que distribuye el tráfico de red en un clúster de servidores para optimizar la utilización, mejorar la capacidad de respuesta y una mayor disponibilidad”[64]. Para este caso, los servidores equivalen a los microservicios.

Según Richardson y Smith, un registro de servicios no es más que “una base de datos de instancias de servicios disponibles”[65].

Existen dos tipos de servicios de descubrimiento, lo cuales se describen a continuación.

[63] RICHARDSON, et. al. *Nginx*. p. 35
[64] Citrix Systems Inc. *What is load balancing?* Consulta: 26 de octubre de 2016.
[65] RICHARDSON, et. al. *Nginx*. p. 35.

3.2.2.1. Descubrimiento del lado del cliente

El cliente es responsable de determinar tanto la ubicación de red de las instancias de los servicios disponibles, así como del balance de carga de las peticiones entre ellos.

Un registro de servicios es consultado por el cliente, el cual “es una base de datos de instancias de servicios disponibles”[66] menciona Richardson. Luego el cliente utiliza un algoritmo de balanceo de carga para seleccionar una instancia de servicio disponible y así poder procesar la petición.

La ubicación de red de una instancia de servicio es registrada, en el registro de servicios cuando esta es inicializada, y removida cuando esta es finalizada. Los registros de instancias de servicios típicamente son actualizados periódicamente.

Un ejemplo de un servicio de descubrimiento de este tipo es *Eureka*, parte de *Spring Cloud Netflix*.

3.2.2.2. Descubrimiento del lado del servidor

Para este tipo de servicio de descubrimiento, el cliente realiza una petición por medio de un balanceador de carga. El balanceador de carga consulta el registro de servicios y re direcciona la petición a una instancia de servicio disponible.

Un ejemplo de un servicio de descubrimiento de este tipo es *Elastic Load Balancer* (*ELB*) de *Amazon Web Services*.

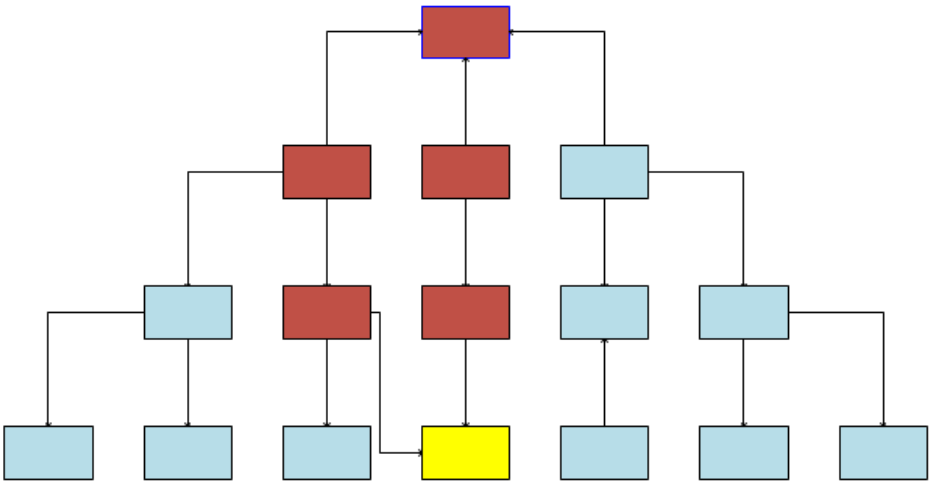
[66] RICHARDSON, et. al. *Nginx*. p. 35.

3.2.3. *Circuit breaker*

Un *circuit breaker* (circuito cerrado), según la empresa *Pivotal Software, Inc.*, sirve para "degradar funcionalidad cuando la llamada a un método falla. Esto permite a un microservicio continuar operando cuando un servicio relacionado falla, previniendo el fallo en cascada y dando tiempo al servicio que fallo de recuperarse"[67].

Cuando un servicio que tiene dependencias falla, se desencadena lo que se conoce como un fallo en cascada, tal como se puede observar en la siguiente figura.

Figura 11. **Fallo en cascada**



Fuente: elaboración propia.

Una solución de software para emplear estos mecanismos es *Hystrix* de *Spring Cloud Netflix*.

[67] Pivotal Software, Inc. *Circuit Breaker.* Consulta: 26 de octubre de 2016.

3.2.4. *Logs* centralizados

En el libro *Professional IIS 7* se definen los *logs* centralizados como “una configuración global de servidores que, escriben todos sus *logs* a un único archivo de registro”[68].

La centralización de *logs*, puede ser de mucha utilidad cuando se trata de identificar los problemas que ocurren en las diferentes instancias de los servicios. También es útil porque se puede.

Rsyslog, *Logstash* y *Kibana* son algunos servicios de *logs* centralizados.

3.2.5. Configuración centralizada

En un sistema distribuido se necesita disponer de un componente de configuración centralizada. Esto permite asegurar que los ficheros de configuración sean únicos para todas las instancias de los microservicios. *Pivotal Software Inc.* Describe un servidor de configuración como “un lugar centralizado para gestionar propiedades externas de aplicaciones alrededor de todos los ambientes”[69].

Este componente lo que hace es tener una copia de los ficheros de configuración, los microservicios se la piden en caliente, es decir siempre que necesitan una propiedad se la piden a este servicio.

Algunos servicios de configuración centralizada son: *Spring Cloud Config*, *Archaius* (parte de *Spring Cloud Netflix*), *Consul*, *Zookeeper*.

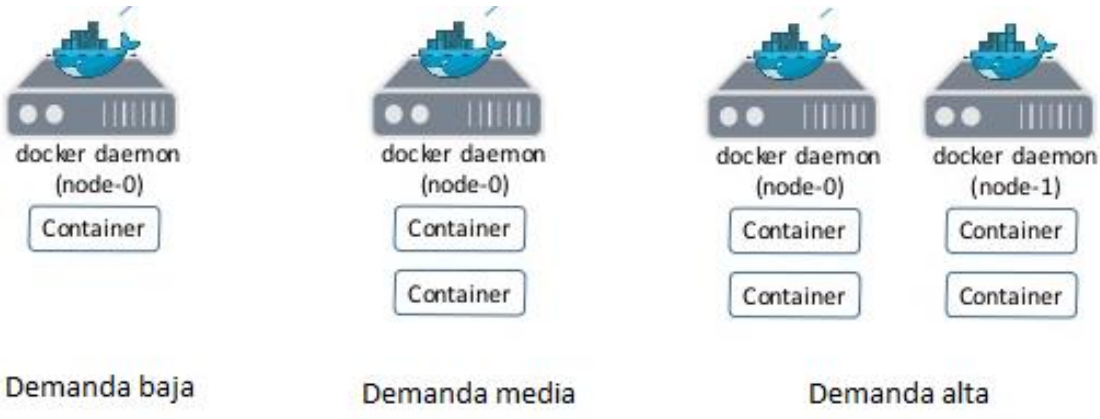
[68] SCHAEFER, et. al. *Professinoal IIS 7*. p. 172.
[69] Pivotal Software, Inc. *Spring Cloud Config.* Consulta: 26 de octubre de 2016.

3.3. Orquestación

Según Richardson, los servicios de orquestación “permiten definir políticas de escalado en función de la demanda”[70].

Por ejemplo, se puede definir una política para cuando se tienen tres tipos de demanda diferentes (baja, media y alta), en la cual, dependiendo del caso, ya sea se levanten nuevas instancias de servicios e incluso nuevos nodos, tal como se puede observar en la siguiente figura.

Figura 12. **Ejemplo de orquestación de microservicios**



Fuente: GARRIDO, Miguel. *Componentes de arquitecturas basadas en microservicios*. Consulta: 10 de agosto de 2016.

Los servicios de orquestación más utilizados en la actualidad son: *Kubernetes*, *Docker Swarm* y *Ribbon* parte de *Spring Cloud Netflix*.

[70] GARRIDO, Miguel. *Componentes de arquitecturas basadas en microservicios*. Consulta: 10 de agosto de 2016.

3.4. Despliegue

Desplegar una aplicación de microservicios es desafiante debido a la complejidad que conlleva por ser una arquitectura distribuida.

Richardson menciona algunos de estos retos en el siguiente enunciado: “Una aplicación de microservicios consiste en muchos servicios. Estos servicios pueden estar escritos en una variedad de lenguajes y *frameworks*. Cada uno tiene requerimientos de despliegue en específico, aún más desafiante es que este despliegue tiene que ser rápido, confiable y efectivo en costos”[71].

A continuación, se describen los tipos de patrones de despliegue.

3.4.1. Múltiples instancias de servicios por *host*

Este es el enfoque común para desplegar aplicaciones de microservicios. Según Richardson al usar este patrón “se disponen de uno o más *hosts* físicos o virtuales y cada uno corre múltiples instancias de servicios”[72].

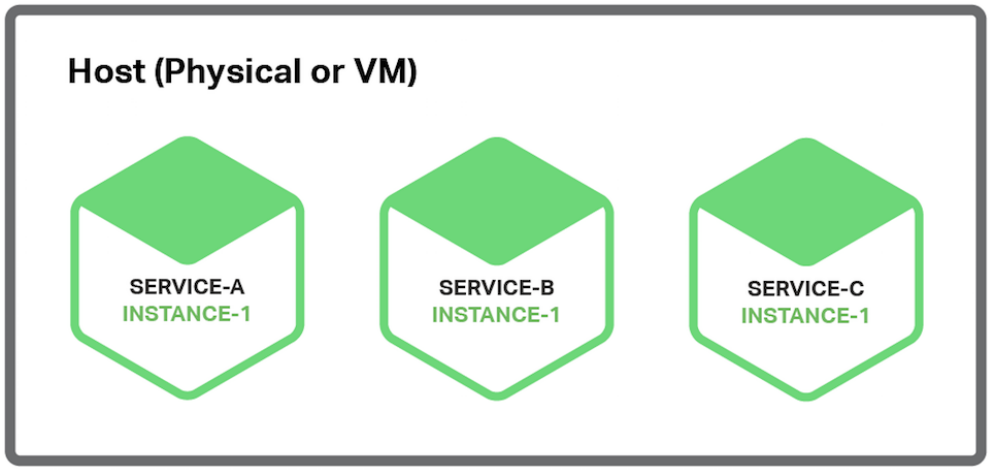
Algunas ventajas que tiene este patrón son: el uso eficiente de los recursos, ya que las instancias de los servicios comparten recursos; el despliegue e inicio de una instancia de servicio es relativamente rápida.

A pesar de sus beneficios, este patrón también presenta algunas desventajas, por ejemplo, no hay aislamiento si múltiples instancias de un servicio corren en el mismo proceso, lo cual, si un servicio falla, puede hacer que los demás servicios fallen.

[71] RICHARDSON, et. al. *Nginx*. p. 55.
[72] Op. Cit. p. 56.

En la siguiente figura puede apreciar como un mismo *host* contiene varios servicios.

Figura 13. **Estructura del patrón de múltiples instancias por *host***



Fuente: RICHARDSON, et. al. *Nginx*. p. 56.

3.4.2. Instancia de servicio por *host*

Al utilizar este patrón se ejecuta cada instancia del servicio de forma aislada en su propio *host*. En seguida, se describen dos tipos de enfoques para este tipo de patrón.

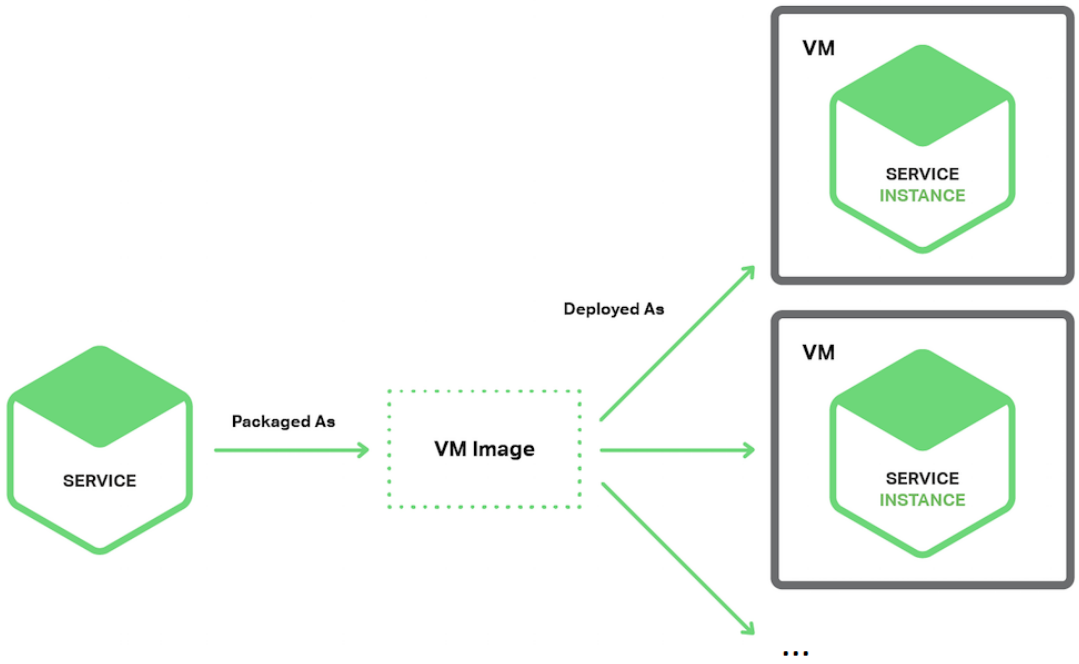
3.4.2.1. Instancia de servicio por máquina virtual

Tal como su nombre lo indica, en este tipo de patrón “se empaqueta cada servicio como una imagen de máquina virtual. Cada instancia de servicio es una máquina virtual que es iniciado usando esa imagen de máquina virtual”[73].

[73] RICHARDSON, et. al. *Nginx*. p. 58.

En la siguiente figura puede apreciar, como un servicio es empaquetado solamente para una máquina virtual.

Figura 14. **Estructura de instancia de servicio por máquina virtual**



Fuente: RICHARDSON, et. al. *Nginx*. p. 58.

Este patrón tiene varios beneficios: cada servicio se ejecuta en completo aislamiento; tiene recursos fijos asignados; se pueden aprovechar los beneficios de la nube; una vez que la aplicación esta empaquetada como imagen de máquina virtual esta queda encapsulada.

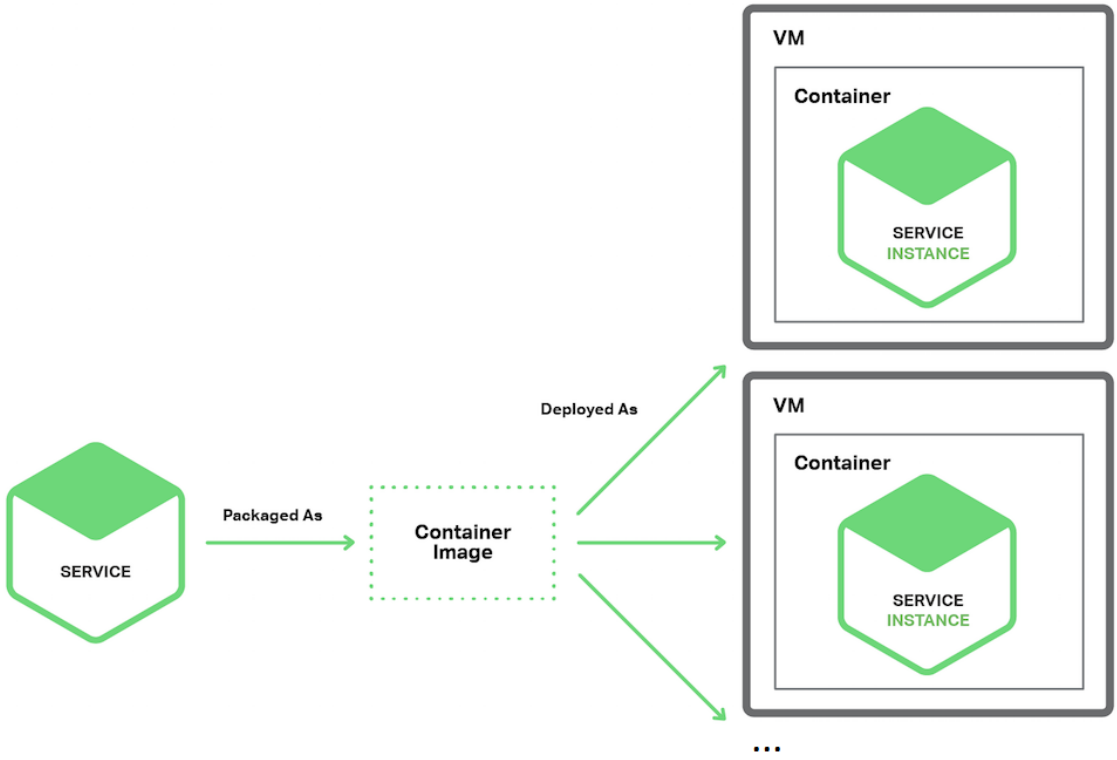
Algunas desventajas que presenta este patrón son: menor eficiencia en la utilización de los recursos; si se utiliza un *IaaS*, típicamente se cobra extra por cada máquina virtual; la construcción, el inicio y el despliegue de la máquina virtual son lentos.

3.4.2.2. Instancia de servicio por contenedor

Para este caso, cada instancia del servicio se ejecuta en un contenedor, lo que implica que se debe empaquetar la aplicación en una imagen de contenedor. Usualmente se ejecutan múltiples contenedores en cada *host*, ya sea físico o virtual.

En la siguiente figura puede apreciar, como un servicio es empaquetado para un contenedor.

Figura 15. **Estructura de instancia de servicio por contenedor**



Fuente: RICHARDSON, et. al. *Nginx*. p. 60.

Los beneficios de este enfoque son similares al enfoque anterior: se aíslan las instancias del servicio; se puede monitorear fácilmente los recursos utilizados por cada contenedor; encapsulan la tecnología utilizada para implementar los servicios; son una tecnología liviana; la construcción, el inicio y el despliegue del contenedor son muy rápidos.

Algunas desventajas que tiene este tipo de enfoque son: la infraestructura para este tipo de tecnología no es tan madura; no son tan seguros como las máquinas virtuales, ya que se comparte el *kernel* del sistema operativo *host* y esto puede ocasionar que, si un contenedor falle, los otros contenedores también lo hagan; al igual que con el enfoque de máquinas virtuales, si se utiliza un *IaaS*, se cobra extra por cada contenedor.

3.5. Principios

Es común en la entrega de software, tener principios que ayuden en este proceso. Cuando se presenta una opción, se observan los principios y se determina si estos son aplicables a una determinada situación. Los principios son opiniones de como las cosas deben ser hechas y restricciones de cómo se deben usar. Muchas organizaciones realizan sus propios principios.

Sam Newman, durante el trabajo que ha realizado con muchas empresas que implementan microservicios, ha establecido una lista de principios[74] que han ayudado a dichas empresas, a construir microservicios exitosos:

- Modelado en torno al dominio del negocio
- Cultura de automatización

[74] NEWMAN, Sam. *The Principles of Microservices: Embrace Autonomy to Optimize Performance*. Consulta: noviembre de 2016.

- Detalles de implementación ocultos
- Descentralización de todas las cosas
- Despliegues independientes
- Consumidor primero
- Aislamiento de fallos
- Altamente observable

3.6. Ventajas y desventajas

Tal como cualquier otro estilo de arquitectura, los microservicios tienen ventajas y desventajas. Estas se deben de entender bien, para hacer la mejor elección. La tabla siguiente muestra algunas ventajas y desventajas que presenta este tipo de arquitectura:

Tabla II. **Listado de ventajas y desventajas de los microservicios**

Ventajas	Desventajas
Tecnología heterogénea	Las pruebas son más complejas
Resistencia	El monitoreo es más complejo
Escalamiento	Complejidad operacional
Alineamiento organizacional	Se utilizan más máquinas *hosts*
Optimización para reemplazos	

Fuente: elaboración propia.

3.7. Casos de uso

Los microservicios han de emplearse cuando la organización, empresa, personas individuales, sea el ente que sea necesite más de alguna de las siguientes características dentro de su desarrollo:

- Flexibilidad tecnológica
- Escalabilidad
- Facilidad de despliegue
- Distribución de responsabilidades
- Reusabilidad de funcionalidades
- Facilidad de reemplazo

Se debe tomar muy en cuenta, que la mayoría de estas características aplica para software que está en constante crecimiento, si es un software el cual no tendrá mucho crecimiento, se podría optar mejor por una arquitectura monolítica.

3.8. Escalabilidad

Una aplicación de microservicios tiene que ser capaz de resistir cargas elevadas, lo que implica que tiene que poder escalar. Según Lucas Krause, autor del libro *Microservices: Patterns and Applications*, la escalabilidad es la "habilidad de un sistema de soportar varias cantidades de tráfico y carga"[75].

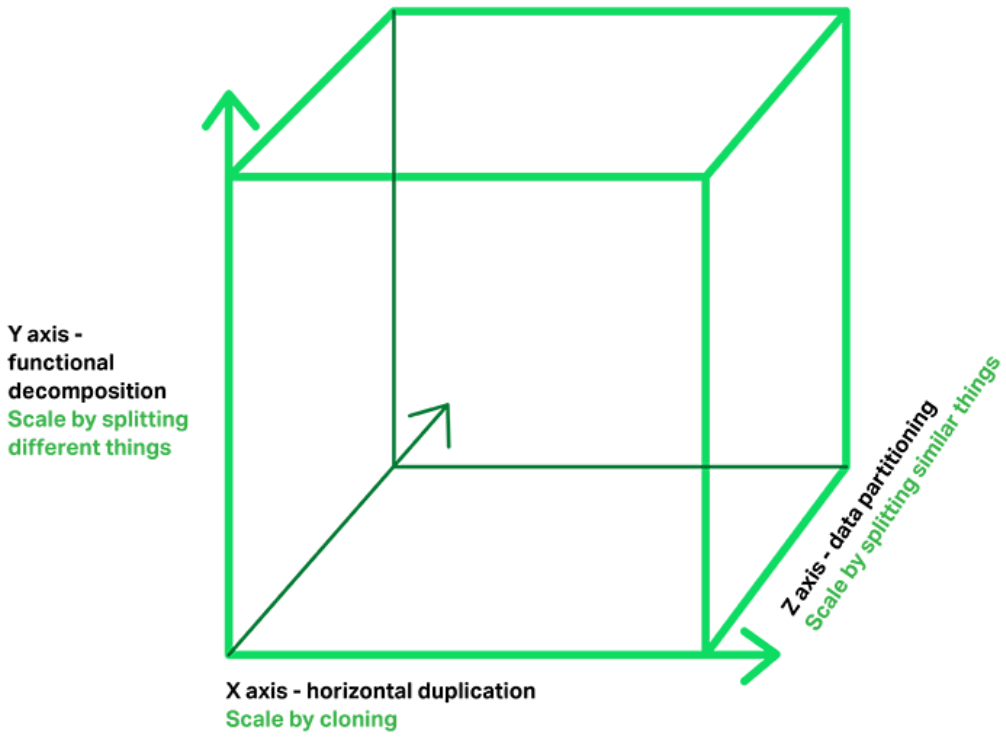
Existe un modelo de escalabilidad de tres dimensiones (eje X, Y y X), llamado el cubo de escalabilidad. Este modelo fue creado por Martin Abbott y Michael Fisher, y presentado en su libro *The Art of Scalability*.

El escalamiento en el eje X, consiste en ejecutar múltiples copias de una aplicación detrás de un balanceador de carga. El escalamiento en el eje Y, consiste en dividir la aplicación en múltiples, diferentes servicios. Cuando se utiliza el escalamiento en el eje Z, cada servidor ejecuta una copia idéntica de la

[75] KRAUSE, Lucas. *Microservices: Patterns and Applications*. p. 20.

aplicación, pero a diferencia del eje X, cada servidor es responsable solo de un subconjunto de datos.

Figura 16. **Cubo de escalabilidad**



Fuente: RICHARDSON, et. al. *Nginx*. p. 6.

Las aplicaciones de microservicios, utilizan los tres tipos de escalabilidad juntos. En el eje Y se descompone la aplicación en microservicios. En tiempo de ejecución, en el eje X se ejecutan múltiples instancias de cada servicio detrás de un balanceador de carga. Algunas aplicaciones usan el eje X al particionar los servicios.

4. IMPLEMENTACIÓN DE MICROSERVICIOS

En este capítulo se procede a realizar la implementación de una arquitectura de microservicios utilizando virtualización por sistema operativo, así como la implementación de una arquitectura monolítica, usando virtualización completa; se muestran las condiciones iniciales de los escenarios planteados, y luego un análisis comparativo de las soluciones.

4.1. Descripción de la aplicación

La aplicación propuesta para realizar el análisis comparativo se describe a continuación:

- Aplicación *web*.
- Se implementó un módulo de gestión de usuarios y un módulo de comentarios.
- Un inicio de sesión como punto de entrada. Tras ingresar credenciales, la aplicación redirecciona a otra página en la cual es posible realizar comentarios. Si las credenciales no son correctas, un mensaje de error es mostrado y se pregunta si se desea crear un usuario.

Esta aplicación se realizará utilizando una arquitectura de microservicios y una arquitectura monolítica, a su vez cada una fue montada utilizando virtualización por sistema operativo y virtualización completa respectivamente.

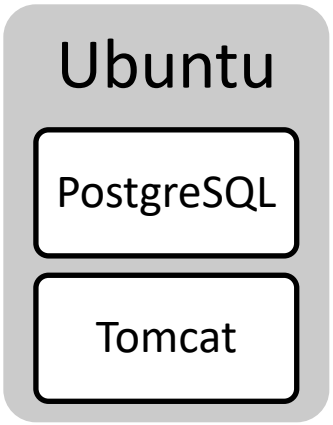
4.2. Escenarios

A continuación, se proporciona una breve descripción de cada escenario implementado, esto incluye su arquitectura y los diferentes componentes empleados.

4.2.1. Escenario 1: arquitectura monolítica y virtualización completa

En la siguiente figura se muestra la arquitectura para el escenario 1. Todos los componentes de la aplicación quedan en un solo paquete: la base de datos, el servidor *web* y la lógica del negocio.

Figura 17. **Arquitectura de escenario 1**



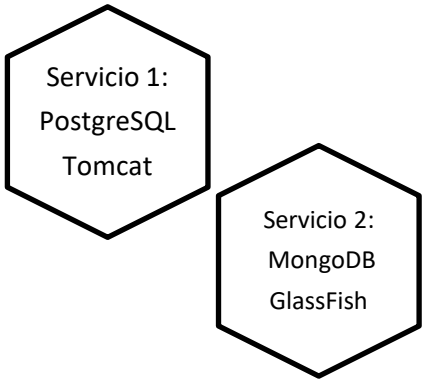
Fuente: elaboración propia.

En este escenario, la aplicación monolítica fue montada en una máquina virtual, utilizando *Virtual Box*.

4.2.2. Escenario 2: arquitectura de microservicios y virtualización por sistema operativo

Para este escenario, la aplicación se descompuso en dos servicios principales, gestión de usuarios y gestión de comentarios. Cada servicio cuenta con su propia base de datos, servidor *web* y su propia lógica de negocio.

Figura 18. **Arquitectura de escenario 2**



Fuente: elaboración propia.

El servicio 1 se implementó usando *Java* como lenguaje de programación, *PosgreSQL* como motor de base de datos y se despliega en un servidor *GlassFish*. Para el servicio 2, se desarrolló igualmente con *Java*, como motor de base de datos *MongoDB* y en *Tomcat* se realizó el despliegue de la aplicación. La figura anterior muestra lo antes descrito.

En este escenario, cada servicio estará montado en un contenedor *Docker* (véase los apéndices).

4.3. Hardware usado

Las especificaciones del hardware utilizado para los escenarios, se describe a continuación:

- Modelo de computador: *Lenovo ThinkPad Edge* 550
- Memoria *RAM*: 12 GB *DDR3*, 798 *MHz*
- Disco duro *SATA* 500 *GB*, 7200 *RPM*
- *Intel Core* i5-5200U, 2.20 *GHz*
- Sistema operativo anfitrión: *Windows 10 Enterprise 64bit*

4.4. Pruebas

Se evaluaron diferentes aspectos a través de distintas estrategias, como por ejemplo la utilización de las herramientas de *benchmark*, *Apache JMeter*, y *Y-cruncher*, encuestas con preguntas para los desarrolladores acerca de qué tipo de arquitectura considera una mejor opción, pruebas de usuario para determinar la satisfacción que este tiene al usar cada escenario.

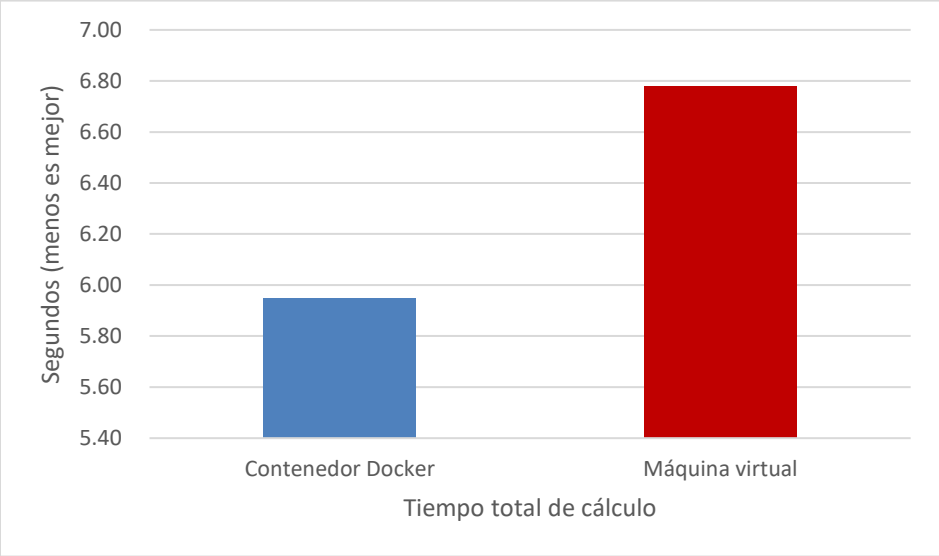
4.4.1. Rendimiento de *CPU*

Y-cruncher es una herramienta de *benchmark* multi-hilo para sistemas multi-núcleo para calcular el valor de Pi.Con esta herramienta se realizaron pruebas de estrés para medir el rendimiento del *CPU*. El tiempo total fue usado para verificar los resultados, y este consiste en el tiempo total de cómputo más el tiempo requerido para elaborar y procesar el resultado.

La figura siguiente muestra los resultados de la herramienta *Y-cruncher*. Como se puede apreciar, la implementación de virtualización por sistema

operativo es más eficiente en el uso del *CPU* en comparación con el uso de virtualización completa, aproximadamente 0,841 segundos de diferencia entre ambas tecnologías.

Figura 19. **Resultados *Y-cruncher***



Fuente: elaboración propia.

Los resultados anteriores, se ven reflejados en la eficiencia multi-núcleo, la cual indica en donde la utilización de contenedores es hace un uso más eficiente alrededor de 3,12 %, tal como se puede apreciar en la siguiente tabla.

Tabla III. **Eficiencia multi-núcleo**

Tecnología	Eficiencia multi-núcleo
Contenedor *Docker*	94,15
Máquina virtual	91,03

Fuente: elaboración propia.

4.4.2. Comparación de tamaño

Para la comparación del tamaño que ocupa cada escenario planteado, se tomó como criterio, el número de instancias que podría almacenar un disco duro de 500 GB, dependiendo de determinadas escalas, en las cuales se encuentra el promedio de una máquina virtual y un contenedor.

Tabla IV. **Tamaños de máquinas virtuales**

Escala	**10 GB**	**20 GB**	**40 GB**
Número de VMs	45	22	11

Fuente: elaboración propia.

Tabla V. **Tamaños de contenedores *Docker***

Escala	**125 MB**	**250 MB**	**500 MB**
Número de contenedores	360	180	90

Fuente: elaboración propia.

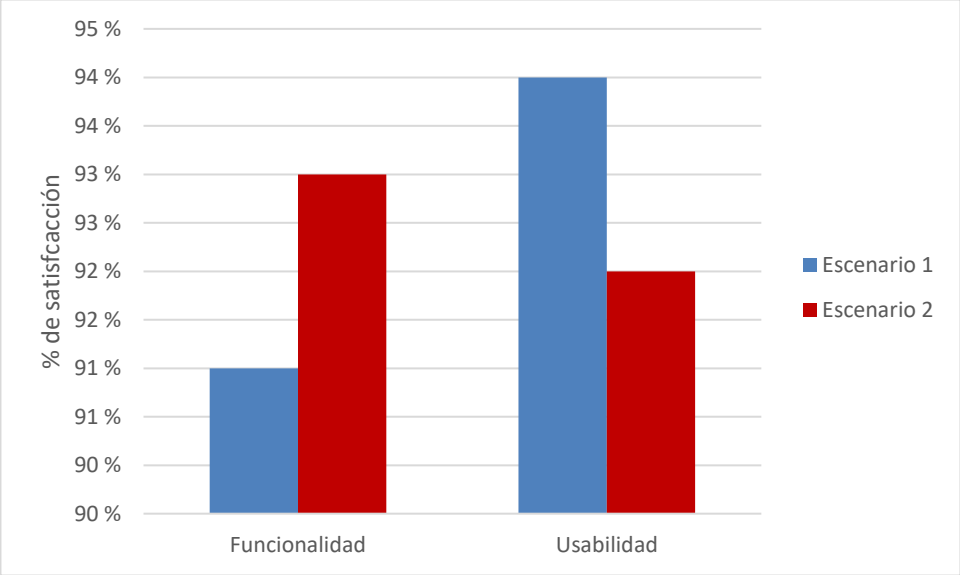
Tal como se puede observar en las tablas anteriores, la aplicación monolítica, al estar en una máquina virtual, dependiendo de que tanto almacenamiento se le haya asignado (por ejemplo 10, 20 y 40 GB), solamente se puede tener un máximo de 45 instancias, en comparación de la aplicación de microservicios que se puede llegar a tener más de 300 contenedores, si el contenedor ocupa alrededor de 125 MB.

4.4.3. Pruebas de funcionalidad y usabilidad

Se realizaron pruebas de usuario para evaluar la funcionalidad y usabilidad, de cada uno de los escenarios planteados.

El usuario, tras la prueba de cada escenario, determinó si la aplicación realiza el trabajo deseado y si esta, generó una sensación de satisfacción. Los resultados de esta prueba se muestran en la figura siguiente.

Figura 20. **Resultados prueba de usuario**



Fuente: elaboración propia.

4.4.4. Rendimiento de aplicaciones

Para analizar y medir el desempeño tanto de la aplicación monolítica como la de microservicios se utilizó la herramienta *Apache JMeter*.

Los resultados arrojados por la herramienta *JMeter* se muestran en la siguiente tabla.

Tabla VI. **Eficiencia multi-núcleo**

Tipo de arquitectura	Promedio (en ms)	Mínimo (en ms)	Máximo (en ms)
Microservicios	150	1	525
Monolítica	163	1	590

Fuente: elaboración propia.

La prueba se realizó enviando 500 peticiones a cada aplicación. El promedio de respuesta fue de 150 y 163 ms, respectivamente para la arquitectura de microservicio y la arquitectura monolítica. El tiempo máximo para responder una petición, para la arquitectura de microservicios fue de 525 ms. Para la arquitectura monolítica, se alcanzó un tiempo máximo para responder una petición de 590 ms. Ambas arquitecturas alcanzaron tuvieron un mínimo de 1 ms para responder a una petición.

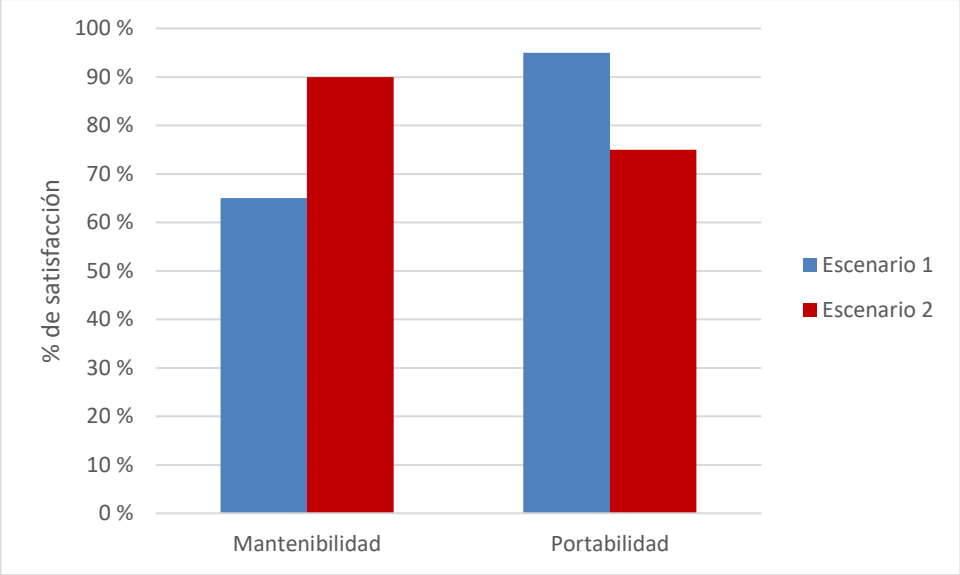
4.4.5. Pruebas de mantenibilidad y portabilidad

Para realizar estas pruebas se recopilaron datos por medio de una encuesta dirigida a desarrolladores y personal del área de operaciones, ya que son estos los que tienen contacto directo con las tecnologías involucradas, virtualización y la arquitectura de software.

Estas encuestas muestran como los desarrolladores y los miembros del área de operaciones perciben la habilidad de ambos escenarios para poder realizar cambios en ellos rápidamente y la habilidad que estos tienen, para correr en diferentes entornos.

Los resultados finales de las encuestas indican efectivamente que la mantenibilidad utilizando una arquitectura de microservicios se ve afecta por ser una arquitectura distribuida, en comparación de una arquitectura monolítica. La portabilidad, según el equipo *DevOps*, se incrementa utilizando virtualización por sistema operativo. En la figura siguiente se puede apreciar los resultados para ambos escenarios.

Figura 21. **Resultados de encuestas a equipos *DevOps***



Fuente: elaboración propia.

4.5. Análisis comparativo

En los resultados se puede observar que las ventajas de una arquitectura de microservicios son bastantes en comparación de una arquitectura monolítica, pero se debe tener en cuenta que el utilizar este enfoque arquitectónico conlleva una serie de problemas, tales como la falta de entendimiento de cómo

funciona el sistema, la resistencia al cambio por parte del equipo de desarrolladores, etc. Es acá en donde entra en juego utilizar *DevOps* para alinear la parte de desarrollo y las de operaciones, para lograr así el mejor resultado empleando este tipo de arquitectura.

Algo a tomar muy en cuenta al elegir entre una arquitectura de microservicios y una monolítica, es el tamaño de la aplicación, ya que no vale la pena utilizar una arquitectura de microservicios en una aplicación que no seguirá creciendo, será más el trabajo que cuesta implementar este tipo de arquitectura a los resultados que se obtendrán, ya que serán relativamente iguales con una arquitectura monolítica.

Ahora bien ¿es conveniente utilizar hipervisores o contenedores? Como se puede observar alrededor de la investigación se mostró las ventajas y desventajas de ambos enfoques, algunas personas podrán pensar en que los contenedores vinieron a sustituir por completo a los hipervisores, pero la realidad es tomar lo mejor de ambos mundos, es decir, combinar hipervisor con contenedor, ya que se estaría obteniendo las ventajas de los hipervisores sumado a las ventajas de los contenedores, complementándolo con una arquitectura de microservicios tener una mayor portabilidad y rapidez en el desarrollo de las aplicaciones y dar una mejor calidad a los usuario.

CONCLUSIONES

1. La mejor opción para implementar el patrón de arquitectura de microservicios es con el uso de contenedores, ya que estos eliminan las dificultades que conlleva el implementar microservicios.

2. Al utilizar contenedores, el tiempo de implementación se mejora considerablemente, tanto al utilizar una arquitectura monolítica como una de microservicios.

3. El implementar una arquitectura de microservicios utilizando virtualización por sistema operativo es el punto de unión para el desarrollo ágil, la entrega continua y la utilización de metodologías como *DevOps*, ya que, al utilizar contenedores, el tiempo de desarrollo y de despliegue se reduce a tal modo que queda perfecto para ser utilizado con metodologías de desarrollo ágil, como por ejemplo *Scrum* o *Extreme Programming (XP)*.

4. Los costos al implementar una arquitectura de microservicio o una monolítica dependerán de qué tan grande sea la aplicación a desarrollar y si se utiliza hipervisores o contenedores.

5. Se debe tener en cuenta los errores que presenta *Docker*, ya que aunque es una tendencia, no es una tecnología tan madura y presenta problemas a tomar muy en cuenta a la hora de implementarlo en un ambiente productivo.

RECOMENDACIONES

1. Analizar siempre el sistema, para determinar si es o no conveniente implementar una arquitectura de microservicios, ya que, si bien tiene muchas ventajas, requiere de tiempo el instruir a todo el equipo de desarrollo a aplicar este tipo de arquitectura.

2. No se debe pensar que los contenedores son la sustitución de los hipervisores, es todo lo contrario, los contenedores se pueden combinar con los hipervisores y así lograr un desempeño muchísimo más alto del que tiene, al implementarlo individualmente.

3. Un enfoque en el cual se logren los mejores resultados, incluye alinear líneas de negocio, profesionales, ejecutivos, socios, proveedores, etc. Entonces una manera de alcanzar esto es con principios de agilidad y eficiencia en el que todos los miembros de la empresa colaboren entre sí, con lo cual es muy recomendable migrar a una arquitectura de microservicios, ya que facilita la utilización de enfoques tales como *DevOps*, ya que *Docker* es una tendencia de tecnología a adoptar como estrategia de negocio.

BIBLIOGRAFÍA

1. ABBOTT, Martin L. y FISHER, Michael T. *The Art of Scalability.* Indiana, USA : Pearson Education, Inc., 2010. 624 p. ISBN: 978-0-13-703042-2.

2. ALEXANDER, Christopher, et. al. *A Pattern Language.* Nueva York, USA : Oxford University Press, 1977. 1171 p. ISBN: 0-19-501919-9.

3. Amazon. *Benefits at a Glance*. [en línea]. <https://aws.amazon.com/application-hosting/benefits/?nc1=h_ls>. [Consulta: 21 de octubre de 2016].

4. ————————. *Types of Cloud Computing.* [en línea]. <https://aws.amazon.com/types-of-cloud-computing/?nc1=h_ls>. [Consulta: 19 de octubre de 2016].

5. ————————. *What is Cloud Computing?* [en línea]. <https://aws.amazon.com/es/what-is-cloud-computing/?nc1=h_ls>. [Consulta: 18 de octubre de 2016].

6. CAMPBELL, Sean y JERONIMO, Michael. [en línea]. <https://software.intel.com/sites/default/files/m/d/4/1/d/8/An_Introduction_to_Virtualization.pdf>. [Consulta: 1 de septiembre de 2016].

7. CAUM, Carl. *Continuous Delivery Vs. Continuous Deployment: What's the Diff?* [en línea]. <https://puppet.com/blog/continuous-delivery-vs-continuous-deployment-what-s-diff>. [Consulta: 22 de octubre de 2016].

8. Cisco Systems. *VPN*. [en línea]. <http://www.cisco.com/c/es_es/solutions/security/virtual-private-network-vpn.html>. [Consulta: 18 de octubre de 2016].

9. Citrix Systems Inc. *What is load balancing?* [en línea]. <https://www.citrix.com/glossary/load-balancing.html>. [Consulta: 26 de octubre de 2016].

10. Docker Inc. *Daemon*. [en línea]. <https://docs.docker.com/engine/reference/commandline/dockerd/>. [Consulta: 24 de octubre de 2016].

11. ————————. *Docker Overview.* [en línea]. <https://docs.docker.com/engine/understanding-docker/>. [Consulta: 24 de Octubre de 2016].

12. GARRIDO, Miguel. *Componentes de arquitecturas basadas en microservicios*. [en línea]. <https://www.youtube.com/watch?v=LjqtaaMJi4U>. [Consulta: 26 de octubre de 2016].

13. GARRIDO, Miguel y GALÁN, Raúl. *Arquitecturas basadas en microservicios.* [en línea]. <https://www.youtube.com/watch?v=2SnWpn1pCOs>. [Consulta: 16 de octubre de 2016].

14. GOLDEN, Bernard. *Virtualization for Dummies.* 3a ed. Indiana, USA : Willey Publishing, Inc., 2011. 62 p. ISBN: 978-0-470-94331-1.

15. IBM. *Direccionamiento con IP virtual*. [en línea]. <http://publib.boulder.ibm.com/html/as400/v4r5/ic2931/info/RZAJWVIProute.html>. [Consulta: 30 de octubre de 2016].

16. Institute of Electrical and Electronics Engineers, Inc. *IEEE Std 1471-2000: Recommended Practice for Architectural Description of Software-Intensive Systems.* 1a ed. Nueva York, USA : Institute of Electrical and Electronics Engineers, Inc., 2000. 23 p. ISBN: 0-7381-2518-0.

17. Intel. *Intel.* [en línea]. <https://01.org/sites/default/files/page/vmscontainers_wp_final.pdf>. [Consulta: 22 de octubre de 2016].

18. KEPES, Ben. *Understanding the Cloud Computing Stack: SaaS, PaaS, IaaS. Rackspace US, Inc.* [en línea]. <http://www.rackspace.com/knowledge_center/sites/default/files/whitepaper_pdf/Understanding-the-Cloud-Computing-Stack.pdf>. [Consulta: 6 de septiembre de 2016].

19. KONRAD, Alex. *How Docker Escaped Near-Death To Become Software's Next Big Thing.* [en línea]. <http://www.forbes.com/sites/alexkonrad/2015/07/01/how-docker-escaped-near-death-to-become-softwares-next-big-thing/#28061e0865e8>. [Consulta: 24 de octubre de 2016].

20. KRAUSE, Lucas. *Microservices: Patterns and Applications.* USA: Lucas Krause, 2015. 126 p. ISBN: 978-0692424278.

21. LEWIS, James y FOWLER, Martin. *Microservices.* [en línea] <http://martinfowler.com/articles/microservices.html>. [Consulta: 7 de septiembre de 2016.]

22. MSV, Janakiram. *How Docker Has Changed the DevOps Game.* [en línea] <http://www.forbes.com/sites/janakirammsv/2016/06/13/how-docker-has-changed-the-devops-game/#e97761b1f43e>. [Consulta: 25 de octubre de 2016].

23. NADAREISHVILI, Irakli, MCLARTY, Matt y AMUNDSEN, Mike. *Microservice Architecture - Aligning Principles, Practices and Culture.* USA : O'Reilly Media Inc., 2016. 146 p. ISBN: 978-1-491-95625-0.

24. NEWMAN, Sam. *The Principles of Microservices: Embrace Autonomy to Optimize Performance.* [en línea]. <http://martinfowler.com/articles/microservices.html>. [Consulta: 7 de septiembre de 2016].

25. Oracle. *Linux Containers (LXC) - Consolidate with Oracle Linux Containers.* [en línea]. <www.oracle.com/us/technologies/linux/lxc-features-1405324.pdf>. [Consulta: 22 de octubre de 2016].

26. Pivotal Software, Inc. *Circuit Breaker*. [en línea]. <https://spring.io/guides/gs/circuit-breaker/>. [Consulta: 26 de octubre de 2016].

27. ————————. *Spring Cloud Config*. [en línea]. <https://cloud.spring.io/spring-cloud-config/>. [Consulta: 26 de octubre de 2016].

28. PORTNOY, Matthew. *Virtualization Essentials.* 1a ed. Indianapolis, USA: John Willey & Sons, Inc., 2012. 286 p. ISBN: 978-1-118-17671-9.

29. POSTA, Cristian. *Microservices for Java Developers - A Hands-On Introduction to Framework & Containers.* California, USA : O'Reilly Media Inc., 2016. 120 p. ISBN: 978-1-491-96308-1.

30. POULTON, Nigel. *Docker Deep Dive*. [en línea] <https://www.pluralsight.com/courses/docker-deep-dive>. [Consulta: 26 de octubre de 2016].

31. Red Hat Inc. *Chapter 1. Introduction to Control Groups (Cgroups). Red Hat.* [en línea]. <https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/6/html/Resource_Management_Guide/ch01.html>. [Consulta: 22 de octubre de 2016].

32. ————————. *Understanding Linux containers. Red Hat.* [en línea]. <https://www.redhat.com/en/insights/containers>. [Consulta: 25 de octubre de 2016].

33. RICHARDSON, Chris y SMITH, Floyd. *Nginx.* [en línea]. <https://www.nginx.com/resources/library/designing-deploying-microservices/?utm_source=microservices-from-design-to-deployment-ebook-nginx&utm_medium=blog>. [Consulta: 26 de octubre de 2016].

34. SCHAEFER, Ken, et. al. *Professinoal IIS 7.* Indianapolis, USA: Wiley Publishing Inc., 2008. 840 p. ISBN: 978-0470097823.

35. SHARMA, Sanjeev. *DevOps para Dummies.* Limited. Hoboken : John Wiley & Sons, Inc., 2014. 63 p. ISBN: 978-1-119-00406-6.

36. SHULTE, Garth. *What is Docker?* [en línea] <https://www.youtube.com/watch?v=aLipr7tTuA4>. [Consulta: 24 de octubre de 2016].

37. STANOEVSKA-SLAVENA, Katarina, Wozniak, Thomas and Ristol, Santy. *Grid and Cloud Computing - A Business Perspective on Technology and Applications.* Berlin, Alemania: Springer, 2010. ISBN: 978-3-642-05192-0.

38. SUSE. *SUSE.* [en línea]. <https://www.suse.com/documentation/sles11/pdfdoc/lxc_quickstart/lxc_quickstart.pdf>. [Consulta: 24 de octubre de 2016].

39. TURNBULL, James. *The Docker Book.* s.l. : James Turnbull, 2014. 161 p. ASIN: B00LRROTI4.

40. VADUVA, Alexandru. *Virtualization*. [en línea]. <https://www.packtpub.com/books/content/virtualization>. [Consulta: 24 de octubre de 2016].

41. VMware, Inc. *Virtualization: How It Works*. [en línea]. <https://www.vmware.com/virtualization/how-it-works>. [Consulta: 30 de agosto de 2016].

42. ————. *Virtualization: Overview*. [en línea]. <https://www.vmware.com/virtualization/overview.html>. [Consulta: 30 de agosto de 2016].

43. Wikipedia.org. *Operating-system-level virtualization.* [en línea]. <https://en.wikipedia.org/wiki/Operating-system-level_virtualization>. [Consulta: 18 de octubre de 2016].

44. WILLIAMS, David E., GARCIA, Juan y CROSBY, Simon. *Virtualization with Xen.* USA: Syngress Publishing, Inc., 2007. 384 p. ISBN: 978-1-59749-167-9.

APÉNDICES

Dockerfiles

A continuación, se presentan los archivos *dockerfile* empleados para la creación de los contenedores de ambas arquitecturas realizadas.

Archivo de los archivos *dockerfile* para la aplicación de microservicios

Dockerfile para el microservicio de usuarios

```
FROM tomcat:latest
MAINTAINER wwwwesh@gmail.com
ADD user_service.war /usr/local/tomcat/webapps
```

Dockerfile para el microservicio de comentarios

```
FROM glassfish:latest
MAINTAINER wwwwesh@gmail.com
ADD comment_service.war /domains/domain1/applications
```

Algunos comandos *Docker* utilizados

Para crea las imagenes *Docker*:

```
# docker build -t wesh/servicio_usuario -f DockerFile .
# docker build -t wesh/servicio_comentario -f DockerFile .
```

Para listar las imágenes:

```
$ docker images
```

Para listar los contenedores que se ejecutan actualmente:

```
$ docker ps
```

Para iniciar las imágenes creadas:

```
$ docker run wesh/servicio_usuario
$ docker run wesh/servicio_comentario
```

Fuente: elaboración propia.