GETX

Microservicios para extracción de texto en imágenes.

Arquitecturas de Software para Computación en la Nube

Marcelo Javier Rodríguez

Leg: 245.221

Profesor: Dr . Ing. Jorge Andrés Diaz Pace

Resumen

Este documento expone las decisiones que guían la construcción de una aplicación que permite extraer el texto relativo a fechas desde imágenes con textura homogénea o simpl. .El objetivo fundamental del informe se dirige a los componentes de un SaaS, enfatizando en sus relaciones, detalles de implementación y eventuales oportunidades de mejora.

1. Motivación

El uso masivo de smartphones ha facilitado y enriquecido variedad de formas de comunicación , simplificación de tareas, registro de actividades,etc ; entre ellas, una actividad generalizada es tomar imágenes de momentos significativos o bien registrar una responsabilidad futura. En tal caso, generalmente la imagen incluye texto en lenguaje natural denotando un evento y/o meramente un escaneo de interés (examen, certificado, cita , etc).

De cara a al problema, describimos una acotada solución para procesar y extraer el texto de una imagen y reconocer el día horario, y registrarlo como alarma , evitando los pasos de lectura y registro (manual o automático). En este contexto, se presenta una solución SaaS mediante una Arquitectura de Microservicios.

2. Arquitectura de Microservicios

Para exponer las características de esta arquitectura comenzaremos con una explicación comparativa, contrastando con la arquitectura monolítica. Las figuras 1 y 2 pretenden evidenciar estas diferencias.

Una arquitectura de sistema monolítico es aquel en el que , independientemente del diseño y/o componentes del sistema, su ejecución sigue un proceso único (y posibles múltiples threads) . Tales características redundan en un único artefacto de software que será desplegado como un todo : jar, war . Cuando la demanda de uso de un sistema monolítico fluctúa, el escalado de esa aplicación se consigue replicando "el todo"(o bien verticalmente) . Es claro que el hardware no será aprovechado (varios módulos de la aplicación pueden no ser demandados). En términos de desarrollo, y por su inherente acoplamiento, esta clase de sistemas también tiene limitaciones respecto a la tecnología de programación, impacto respecto a cambio de requerimientos, testing, etc .

En contraste, una arquitectura basada en microservicios confina los componentes de un monolítico como artefactos independientes, con responsabilidad específica (alta cohesión por módulo), y que se comunican mediante una red. Esta independencia se manifiesta superando las limitaciones comentadas para los sistemas monolítico : sólo el componente más demandado puede escalarse en forma individual, esto es escalar horizontalmente, haciendo uso eficiente del hardware subyacente, redundando en mejor performance (considérese latencia de comunicación y computación a gran escala si el dataset/procesamiento es divisible), y réplicas de componentes que brindan mayor disponibilidad; en términos de desarrollo, el testeo por unidad de cada servicio es más simple que probar un todo (no así la integración), aunque su programación puede ser más compleja dada las comunicaciones y exposición de interfaces; también, se trascienden las limitaciones respecto a una tecnología específica.

Como consecuencia, este tipo de arquitecturas resulta atractiva en dominios donde la computación es demandante y variable, al igual que la dinámica de cambio de requerimientos, también cuando la resiliencia y otros atributos de calidad de una arquitectura de sistemas impacta implícitamente en el auge y/ o competitividad de los negocios actuales.

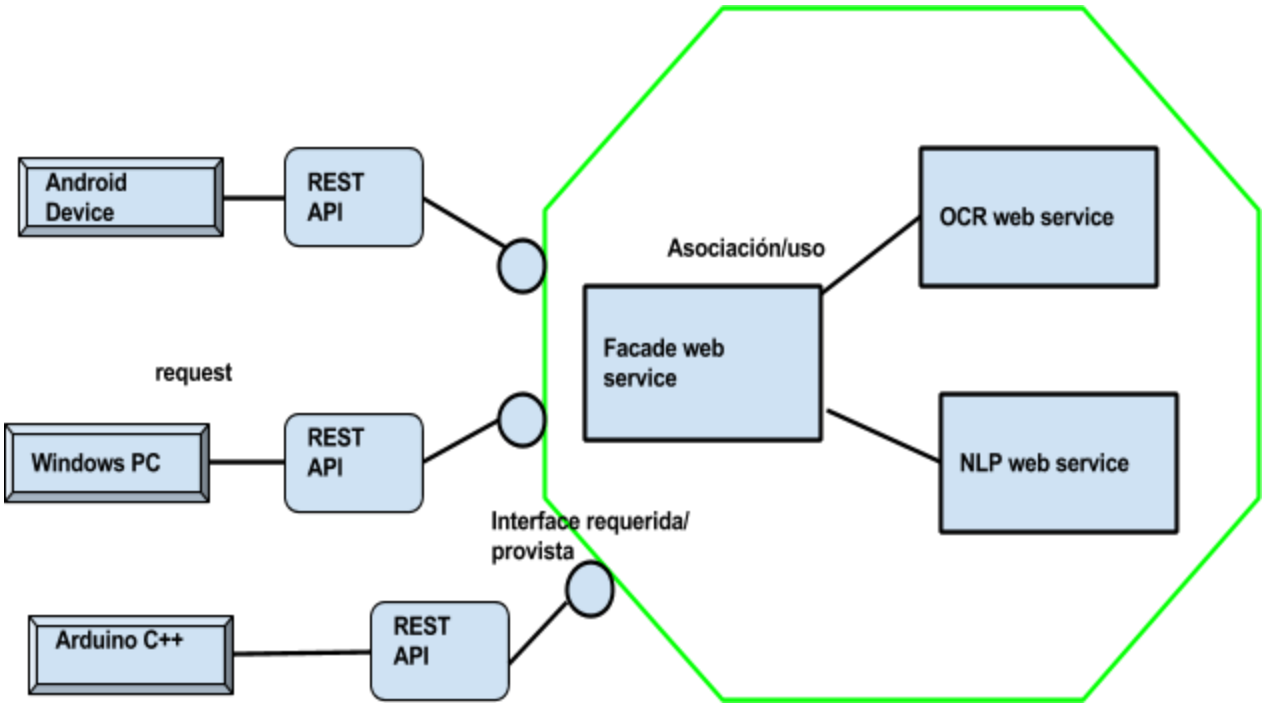

Figura 1. Arquitectura de un sistema monolítico

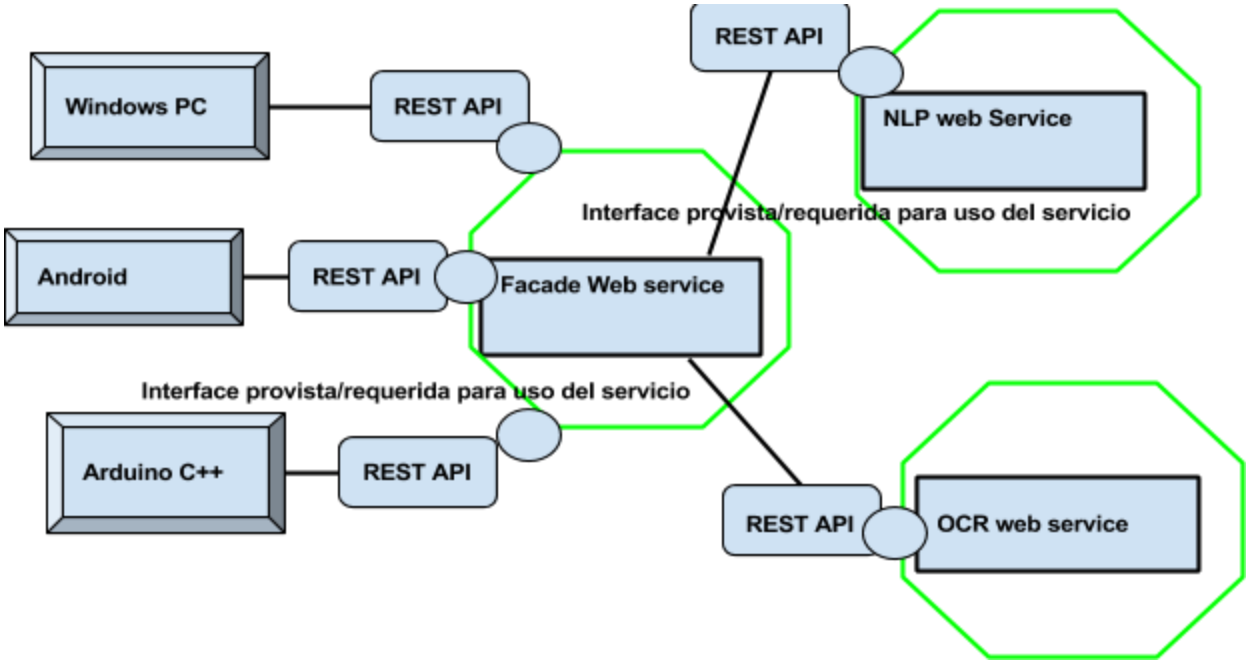

Figura 2.Arquitectura de sistema basada en microservicios

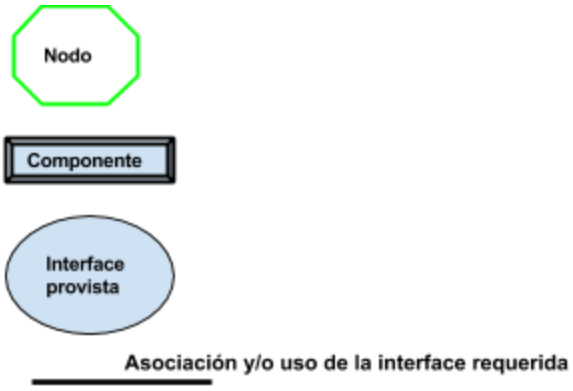

Figura 3. Leyenda .

2.1. Microservicios y Containers

Los containers son procesos normales, que además incluyen todas las dependencias para ejecutar una aplicación; por lo tanto, su objetivo es aislar un proceso (vía namespaces) y hacerlo independiente del entorno de ejecución. En este sentido pueden ser comparables con las máquinas virtuales, pero a diferencia de estas, no emulan un sistema operativo , pues son meramente programas y por lo tanto son más ligeros que las máquinas virtuales ya que no incurren en el overhead impuesto por las capas de virtualización ante eventuales system calls. Como tales, los containers son un medio que encaja perfecto para la implementación y escenarios de ejecución de una arquitectura de sistemas distribuidos .

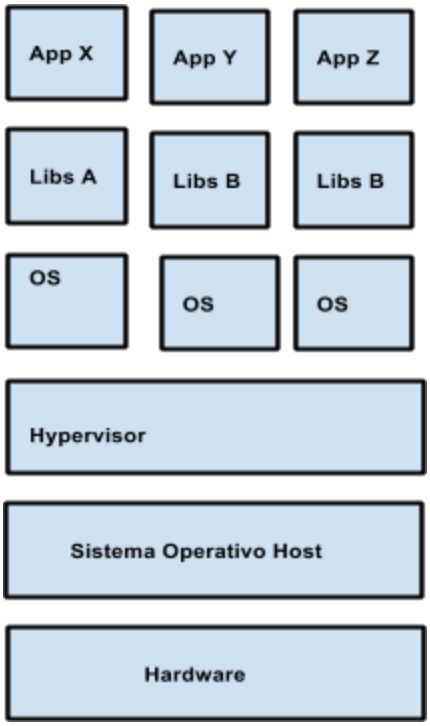

Figura 4. Arquitectura Layers típica de una Máquina Virtual con aplicaciones

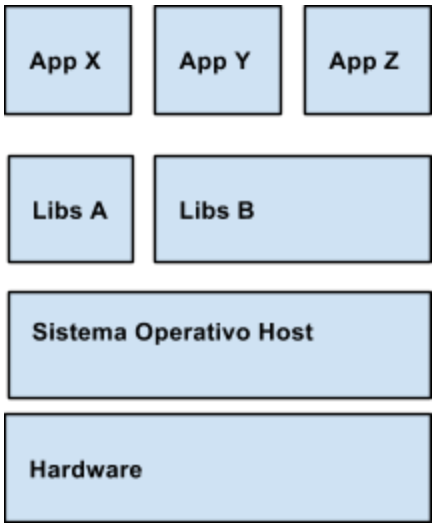

Figura 5.Arquitectura de procesos de un sistema con containers.

En la figuras 3 y 4 se ven tres aplicaciones independientes desplegadas en VMs separadas en un mismo host , y las mismas aplicaciones en un sistema "contenerizado" . El hypervisor es requerido para crear y ejecutar las máquinas virtuales y controlar el acceso al SO. Cada máquina requiere una copia completa del SO. La figura 4 muestra un sistema con containers: las aplicaciones tienen sus propias copias de librerías . El sistema Docker, es el responsable de gestionar los containers similarmente al hypervisor .

2.2. Orquestación de Clusters de containers

Una arquitectura de sistemas distribuidos, específicamente basada en microservicios, puede implementarse utilizando un conjunto/cluster de containers . En lo que sigue se introducirán las tecnologías utilizadas para construir GETX. Docker es un software single-host que le permite crear y compartir imágenes de aplicaciones y ejecutar sus correspondientes containers, y monitorizar su estado. La arquitectura de Docker se presenta en la siguiente figura.

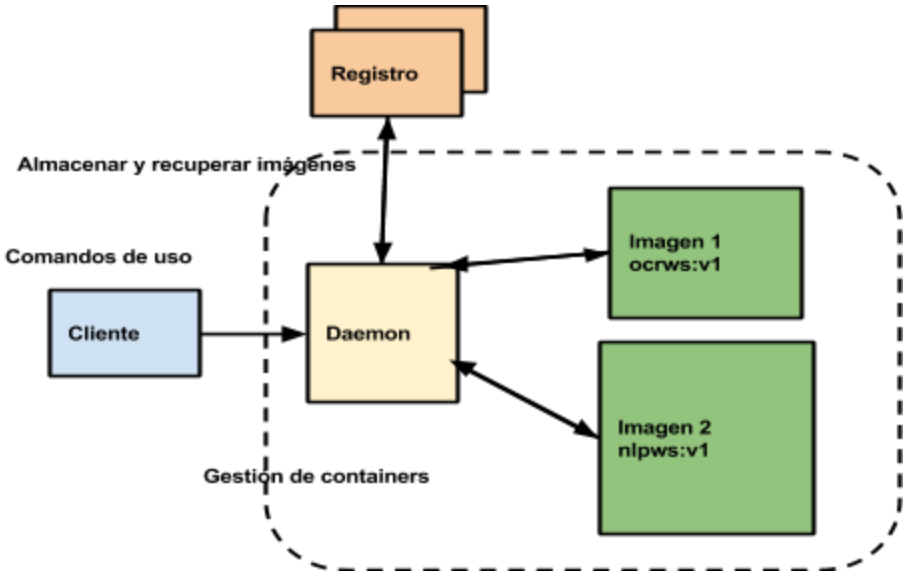

Figura 6. Arquitectura de Docker. Al centro de la arquitectura existe un proceso daemon, responsable de construir, guardar y recuperar imágenes en los registros local y oficial en cloud - Docker hub , y revisar la salud de los containers. El cliente Docker se comunica con el API de Docker usando el protocolo HTTP.

En un ambiente cloud tendremos clusters de host o máquinas virtuales. Estas serán los nodos que hostean los containers que ejecutan los servicios del sistema. En este contexto, eventualmente necesitemos monitorizar el estado de tales containers, realizar actualizaciones de sus imágenes , responder a demandas de tráfico disparando réplicas de nuestros servicios . En consecuencia, es indudablemente más cómodo tratar estos clusters en un nivel de abstracción más alto que facilite describir esta serie componentes como una única entidad. Las herramientas que facilitan esta tarea se denominan Herramientas de Orquestación de/y Clustering . Entonces, "la orquestación" es un forma de tratar todo un cluster como si de un único servicio se tratase . Para lograr esto, un requerimiento implícito para estas herramientas es el descubrimiento de servicios y scheduling/programación de los recursos que están gestionando en función de sus características: una instancia de este problema es el despliegue de los containers cuando hay autoescalamiento, o la migración de un container ante la baja de un nodo considerando dónde hostearlo según el estado del destino . Docker Swarm es la herramienta de orquestación utilizada en este caso particular.

2.3 Docker Swarm

Docker Swarm es una herramienta basada en el API de Docker, orientada a la orquestación de containers desplegados sobre nodos físicos o virtuales. La arquitectura de Swarm ejecuta un proceso agente en el nodo master, este es el responsable de la orquestación; los nodos esclavos envían métricas al agente mediante un mecanismo de heartbeat junto con el token de identificación correspondiente al cluster; con esto, también lograr la tarea de descubrimiento de servicios[1] y el cambio dinámico de los nodos en el cluster. La lista de nodos del cluster actualizada se mantiene en el registro Docker Hub.

[1] Es el proceso de proveer a un cliente de un servicio con la información necesaria para acceder a una instancia del mismo.

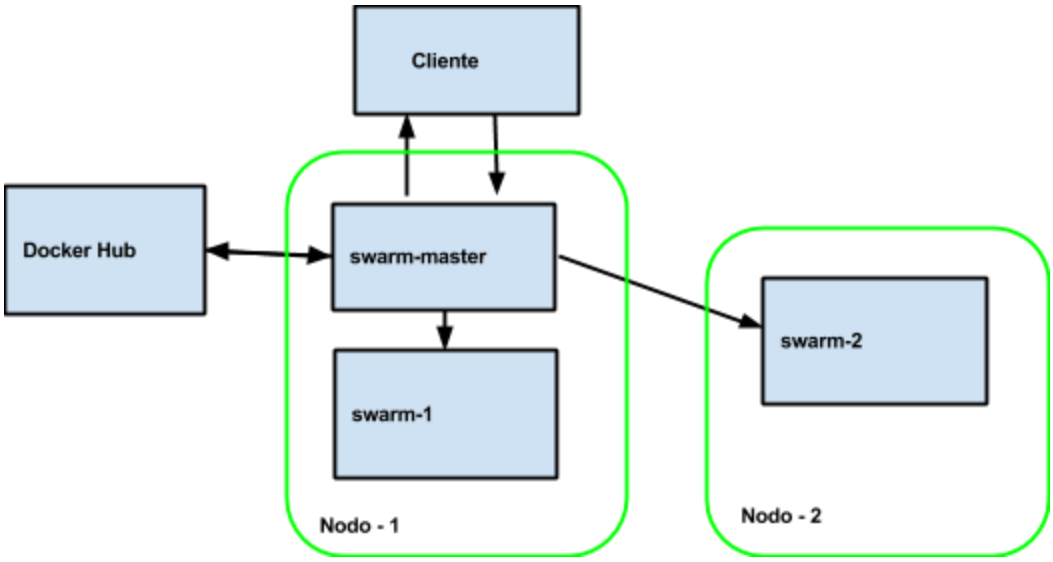

Figura 7. Arquitectura de Docker Swarm

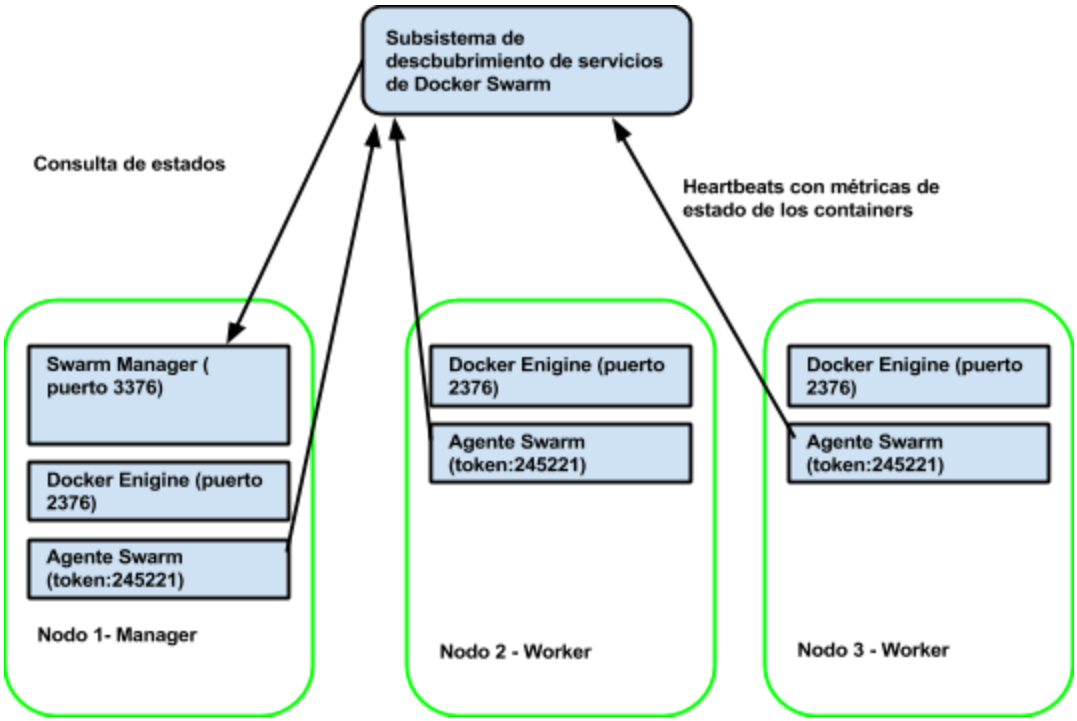

Figura 8.Interacción entre el Manager y Agentes para descubrimiento de servicios . Los agentes Swarm en cada nodo van empujando al subsistema de descubrimiento sus estadísticas de estado y membresía al cluster. El Swarm manager recupera de este subsistema los miembros actuales para ejecutar sus políticas de scheduling de servicios.

La creación de nodos virtuales locales o en una cloud se realiza utilizando Docker Machine y especificando el driver que determina el nodo virtual a crear ; en este caso particular se utiliza Virtualbox . Cada nodo creado con Docker Machine es automáticamente aprovisionado con Docker y puede ser configurado en términos de recursos requeridos y disponibles (hdd, ram, etc).

3. Descripción GETX

Hasta aquí hemos descrito las características relevantes de una arquitectura de microservicios, su relación con los containers y las tecnologías utilizadas para la implementación de nuestro caso . A continuación, profundizaremos en otros aspectos propios y comunes de estas arquitecturas ajustándose al caso de GETX .

3 . 1 Escenario de uso .

Un cliente toma una imagen , realiza un post con la misma a GETX con intención de obtener el evento capturado en la misma. El sistema debe responder en un tiempo razonable para evitar tiempos ociosos de cliente .

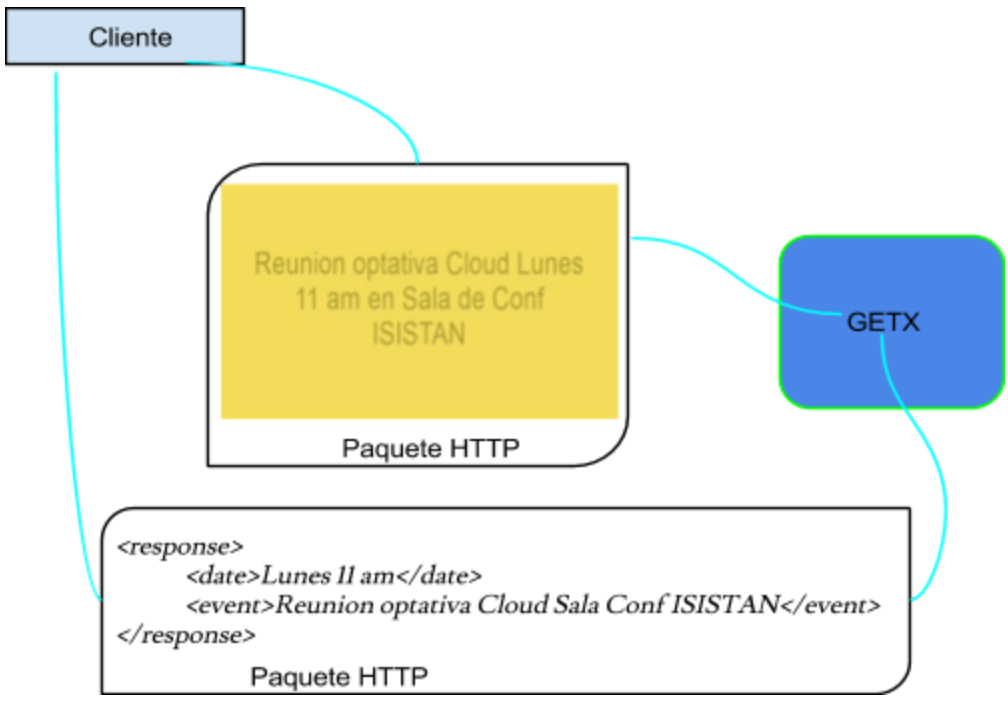

Figura 9. Flujo de trabajo de muy alto nivel del escenario de uso de GETX.

3.2 Deployment de los microservicios de GETX y componentes típicos de Arquitecturas de Microservicios.

En este caso particular,dados los recursos On-Premise , los servicios son desplegados como múltiples instancias de containers Docker por host virtual, de esta manera un nodo virtual es compartido entre varios containers utilizando el mecanismo **mapeo de puertos** entre el host y el puerto expuesto por el container.

En un ambiente cloud las instancias de servicio cambian, y obviamente, también sus direcciones en forma dinámica (fallas,actualizaciones, escalamiento) . Generalmente , y para satisfacer requerimientos no funcionales de sistemas, es *muy común* establecer frente de los servicios un balanceador de cargas. Este actúa como proxy de la aplicación . El cliente de la aplicación se conecta vía direcciones DNS a este intermediario . Las request emitidas por el cliente son adelantadas por el proxy a los servicios del backend en base al nombre del método HTTP y/o headers dentro del paquete. Este balanceador de cargas es consciente del pool de servicios del backend en todo momento, y reacciona a eventos de escalado horizontal up o down. Los servicios pueden auto registrarse mediante un heartbeat; este método de registrado tiene la ventaja de no requerir componentes externos, pero mantiene acoplada la instancia de servicio al registro. Este es el caso con Docker Swarm . El otro enfoque, que desacopla al servicio del registro, es un componente adicional que chequee , o se suscriba a los eventos del pool de instancias de servicios y mantenga el registro.

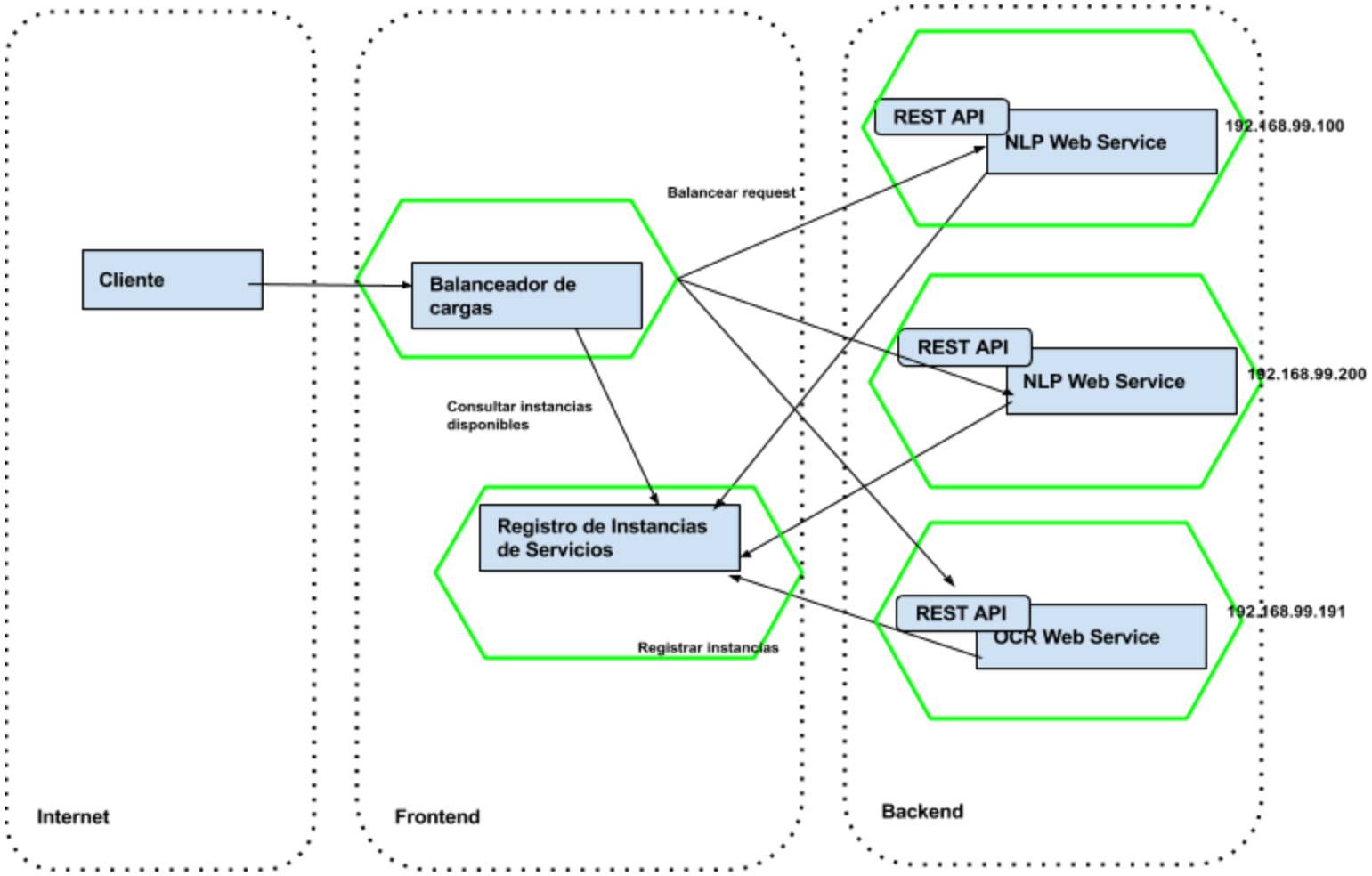

Figura 7. Despliegue del sistema GETX. El balanceador de cargas accede al registro para conocer la dirección de red de los servicios actuales.

En la figura 7 aparecen los componentes que forman la arquitectura de GETX : nodos virtuales, servicios en containers accedidos mediante interfaces REST, un balanceador de cargas y el registro de servicios el cuál está dentro de la herramienta de orquestación Docker Swarm .
Los Web Services que dan la funcionalidad son: OCRWS, implementando la extracción de caracteres desde una imágen; y NLPWS, que reconoce frases con día horario.

3.3 Acceso al servicio de GETX: API REST :

- **Método** : POST
- **URL base:** www.getx.com
- **Descripción:** procesa una imagen para extraer texto
- **Parámetros de entrada** : archivo de imagen
- **Formato de respuesta:** json con texto extraído , nombre del archivo procesado, nombre del servicio e identificación del container que hostea el servicio.

- **Método:** GET
- **URL base:** www.getx.com
- **Descripción:** extrae el texto referente a un día horario.
- **Parámetros de entrada:** texto corto (debe contener fechas en lenguaje natural)
- **Formato de respuesta:** json[2] con texto de fecha, evento descrito, nombre del servicio e identificación del container que hostea el servicio.

[2] **JSON**, acrónimo de *JavaScript Object Notation*, es un formato de texto ligero para el intercambio de datos. JSON es un subconjunto de la notación literal de objetos de JavaScript aunque hoy, debido a su amplia adopción como alternativa a XML, se considera un formato de lenguaje independiente [https://es.wikipedia.org/wiki/JSON]

4. Mejoras propuestas.

La siguiente es una lista de características que se consideran necesarias para dotar al proyecto actual con caracter comercial-profesional.

- **Service Agent:** Incorporar un agente sensible a los eventos que recibe el load balancer para realizar controles de acceso y reunir métricas de uso.

- **Availability 24x7 :** utilizando las mismas tecnologías, sería deseable realizar el deployment en un proveedor de **IaaS** . Docker Machine provee drivers para AWS, Google Compute Engine, ***Digital Ocean*** y Microsoft Azure.
  Como ejemplo, crear un nodo usando la infraestructura de Digital Ocean y ver su dirección IP asignada y archivo de configuración.

  $ docker-machine create --driver digitalocean --*digitalocean*-access-token 245221 machine_name
  $docker-machine env machine_name

- **Service Level Agreement**: asumiendo el cumplimiento de todas mejoras de esta sección, y a mediante pruebas en distintos escenarios y análisis de respuesta de calidad del servicio, construir un documento que describa fehacientemente el nivel que el servicio garantizarse.

- **Broker multi device :** desarrollar distintos adaptadores para proveer interoperabilidad e integración con variedad de clientes.

- **Pay per use Monitor (como Service Agent):** es un interceptor que loguea el uso del servicio. Hay casos que se loguea la salida del servicio (podría querer conocer el tiempo de uso).

- **State database Management:** persiste el estado de los servicios (stateless). Esto promueve la replicación (disponibilidad del sistema).

- **Automatic Scaling** : desarrollar un servicio que monitoriza el estado de carga de trabajo del cluster; Datadog es una herramienta que aplica a esta cuestión. Como solución directa y sencilla, puede construirse un servicio en base a los comandos:

  docker ssh manager "docker service ps <nombre_servicio>
  docker ssh manager "docker node ps <nodo>" , y
  docker ssh <nodo> docker stats $(docker ps --format={{.Names}})

5. Conclusión.

Hasta aquí tenemos una versión básica del sistema GETX . Es una versión simple, pero tiene la funcionalidad suficiente para su objetivo . En consecuencia, el desarrollo del proyecto brindo una experiencia que permitió apreciar varias características de trabajar con containers (y sus imágenes), y como estos conducen a grupos que conforman servicios; entre estas, el descubrimiento de servicios se vuelve una característica esencial para un sistema distribuido y dinámico: los containers y los servicios están en un flujo continuo siendo frenados, ejecutados, etc . Para satisfacer este requerimiento y todo lo relacionado a orquestación de containers, se experimentó con Minikube (versión de cluster de un nodo VM local Kubernetes) , que en su versión actual no dispone de métricas en tiempo real (limitando la certeza para escalar un servicio) y además de estar restringido a un único nodo, fueron los aspectos decisivos para decantar por Docker Swarm .

Es notable que el espacio de los containers es aún joven, pero hay muchas herramientas alrededor. La mejor opción entre ellas será dependiente de la complejidad del proyecto y el esfuerzo a invertir en su desarrollo: particularmente, para GETX, su primer versión utilizaba Docker, y la actual, Swarm (esto da más capacidad de procesamiento si se utilizan nodos en un proveedor de IaaS -si proyectamos el sistema pensando en muchos usuarios-), que sólo implica un leve esfuerzo de aprendizaje.

En lo personal, puedo afirmar que la incursión en el área de la Computación en Cloud, me ha provisto de conceptos en tecnologías nuevas y progresivamente crecientes en la informática. Por último destaco que he sido consciente de cómo las características de una arquitectura de software pueden impactar en la calidad de la solución de un problema.

Apéndice .

a.Repositorio y Documentación.

- **Source Code**
- **Video**.

b.Archivos de ejecución.

Se describen los scripts que contienen la funcionalidad para crear los containers y el cluster para la aplicación.
Nota: Se listan en orden de uso.

Nombre	Descripción	Ejecución
make_getx.sh	Crea el cluster: nodos y despliega los containers descargando las imágenes desde un repositorio Docker Hub. Instala el proxy frente a los servicios. **Nota:** demora hasta 60' hasta tener el servicio listo (en el entorno local)	**$> sh** make_getx.sh
test.sh	Ejecuta el cliente getx.jar que aporta una serie de imágenes predefinidas para apreciar el comportamiento del sistema.	**$> ./**test.sh
delete_getx.sh	Frena los nodos virtuales y los elimina.	**$> sh** delete_getx.sh
getx.jar	Cliente que consume el SaaS GETX	**$> java -jar getx.jar <image_file> <ocrws> <nlpws>**

c. Ejecución.

$> git clone https://github.com/buhtigexa/getx.git && cd getx && sh make_getx.sh

Nota Importante : requiere tener instalado Docker y Docker Swarm .