



Universidad de Oviedo

Disposición (Allocation)



ARQUITECTURA
DEL SOFTWARE

Curso 2022/2023

Jose Emilio Labra Gayo

Disposición

Relación del software con el entorno

¿Dónde se ejecuta cada componente?

¿Infraestructura?

¿Despliegue?



Disposición

Empaquetamiento, distribución, despliegue

Opciones de ejecución

Entornos de ejecución

Tubería de despliegue, despliegue continuo

Software en producción

Patrones para software en producción

Pruebas de software en producción

Logging & Monitorización

Incidentes y post-mortem

Ingeniería del caos

Empaquetamiento, distribución y despliegue

Empaquetamiento

Crear ejecutable a partir del código fuente

Consiste en:

Código compilado

Incluso para lenguajes interpretados (Javascript):

Transpiled, ofuscado & minimizado

Ficheros de configuración

Variables de entorno

Credenciales, etc.

Librerías & dependencias

Manuales de usuario y documentación

Scripts de instalación



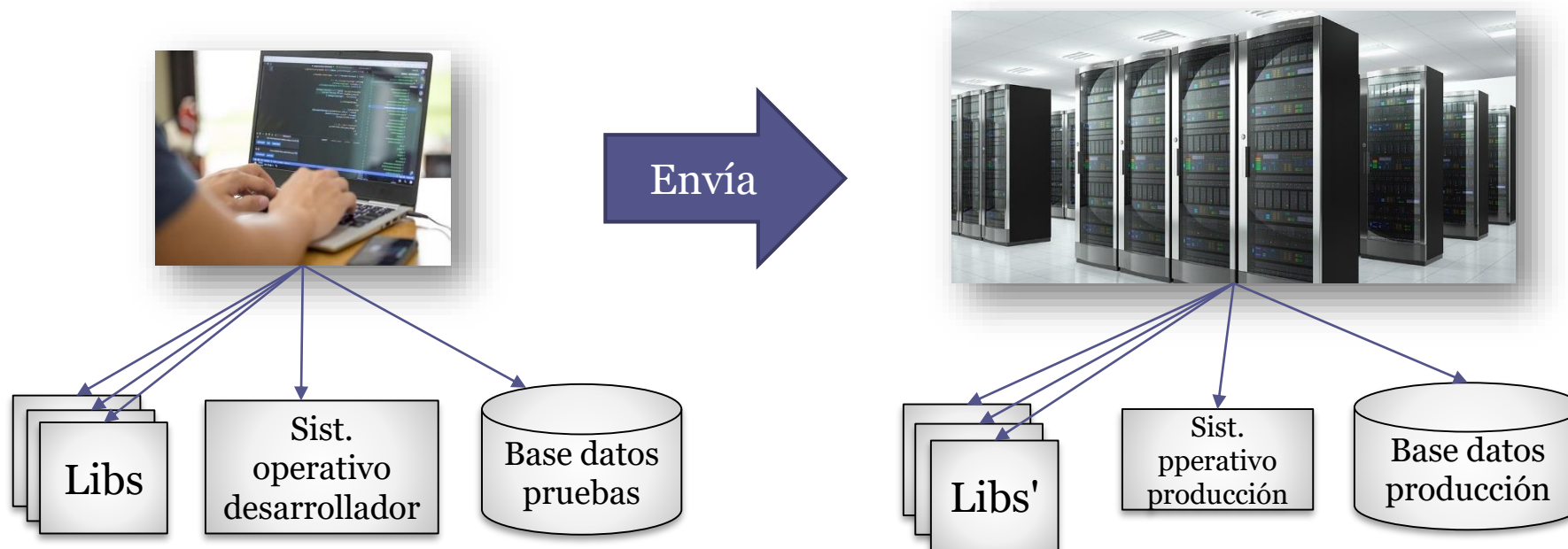
El problema de enviar software

La mayoría del software no es autónomo

Muchas dependencias

Librerías, librerías compartidas, sist. operativo...

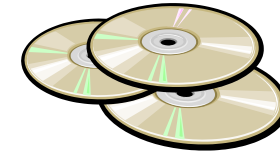
Entorno desarrollo \neq Entorno de producción



Canales de distribución

Distribución tradicional

CDs, DVDs, ...



Basada en Web

Descargas, FTP, ...



Mercados de aplicación

Paquetes Linux

Almacenes de aplicaciones

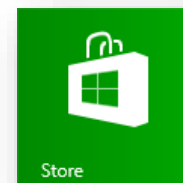
AppStore,

Google Play,

Windows Store



App Store



Store



Google play

Despliegue

Punto de vista de despliegue

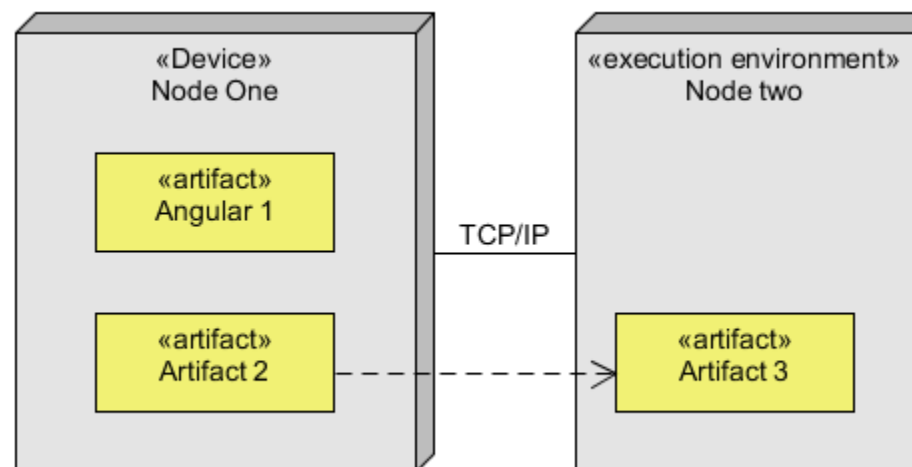
UML proporciona diagramas de despliegue

Artefactos asociados con nodos computacionales

2 tipos de nodos:

Nodo dispositivo (Device)

Nodo de entorno de ejecución



Opciones de ejecución de software

Centro de datos (*On-premises*)

Cloud computing

Edge computing

Fog computing

Computación *on premises*

Software se ejecuta en el edificio

Instalaciones del cliente/centro de datos

Ventajas

Más control del hardware

Actualizaciones, personalización

Seguridad

Cuando está bien configurado

Retos

Requiere inversión de hardware

¿Qué hardware se necesita?

¿Retorno de la inversión?

Costes mantenimiento

También de licencias, espacios,...

Habilidades de administración de sistemas



Computación en la nube

Recursos computacionales bajo demanda

Software as a service (SaaS)

Ventajas

- No requiere inversión inicial
- Menos caro
- Acceso a hardware caro
- No se requieren habilidades de administración de sistemas

Retos

- Seguridad
- Dependencia proveedores de la nube
- Costes variables (posibles sorpresas)
- Requiere habilidades configuración



Metáfora mascota (pet) vs ganado (cattle)

Hace tiempo tratábamos a nuestros servidores como mascotas, por ejemplo Bob el servidor de correo. Si Bob se cae, todo el mundo se pone a arreglarlo. El predisenante no puede leer su correo y es el fin del mundo.

En la nueva forma, los servidores son numerados, como si fuesen Ganado. Por ejemplo, www001 a www100. Cuando un servidor se cae, se lleva a parte, se dispara y se sustituye por otro..



Servidor
"mascota"

- Único e irremplazable
- Gestionado mediante GUI
- Gestión artesanal
- Reservado para nosotros
- Escalado hacia arriba
- ...



Servidores
"Ganado"

- Desechable, uno entre muchos
- Manejado mediante API
- Automatizado
- Bajo demanda
- Escalado hacia afuera
- ...

Más información: <http://cloudscaling.com/blog/cloud-computing/the-history-of-pets-vs-cattle/>

Edge computing

Computación realizada en los dispositivos del usuario

Los dispositivos conectados procesan datos más cerca de donde se crean los datos

Ejemplo: IOT, Coches conectados, ...

Ventajas

- Respuesta más rápida (tiempo real)
- Almacenamiento de datos Micro
- Visualización en las instalaciones
- Independencia (no requiere red)

Retos

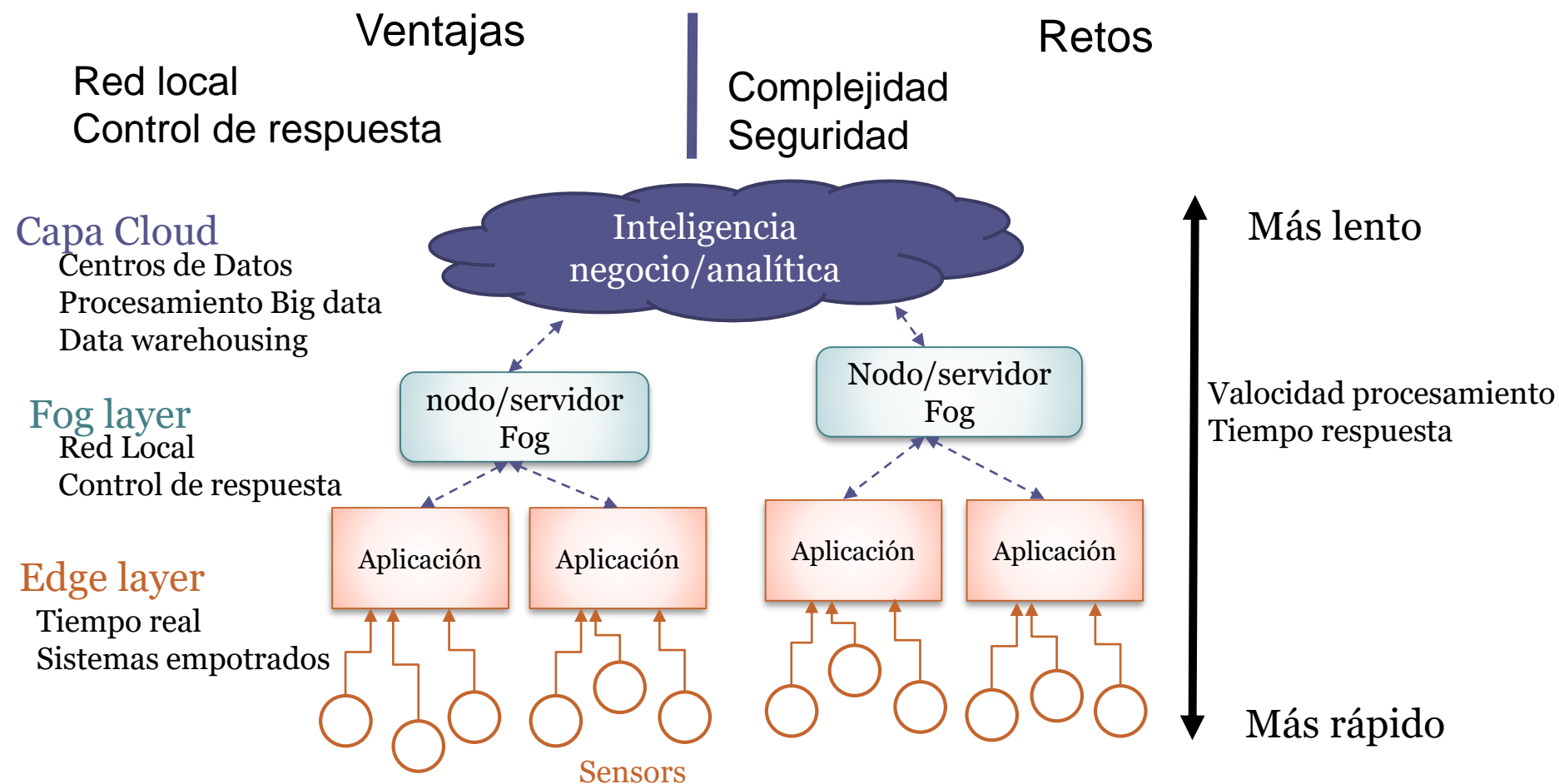
- Menos poder computacional
- No puede acceder a datos necesarios
- Desarrollo de sistemas empotrados



Fog computing

Computación en nodos intermedios

Red área local



Entornos de ejecución

¿Dónde se ejecutará el software?

¿Qué dependencias tiene?

Sistema operativo

Librerías compartidas

Varias opciones

Máquinas físicas

Máquinas virtuales

Contenedores



Máquinas físicas

Múltiples posibilidades

Ordenador convencional

Super-computadores

Granjas de servidores

Dispositivos usuario final



Supercomputador Marenostrum 4(2017)

Fuente: Wikipedia

Ventajas

Control
Rendimiento

Retos

Fiabilidad
Portabilidad

Máquinas virtuales de sistema

Emulación aislada de máquina real

Emulador hardware virtual

Permiten ejecutar múltiples sistemas operativos en una máquina

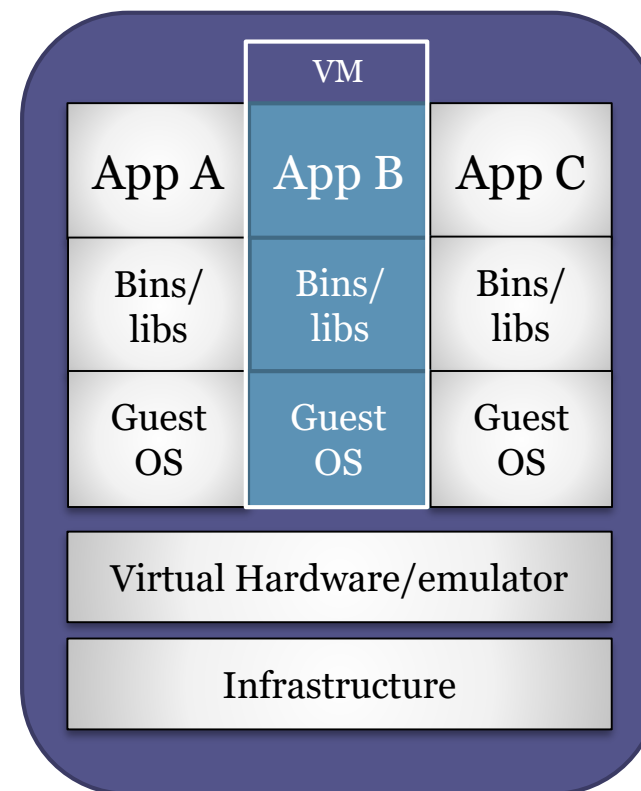
Ejemplos: VMWare, Virtualbox, ...



Máquinas virtuales

Ejecución de aplicaciones en máquinas virtuales

Requieren sistema operativo + librerías



Ventajas

- Portabilidad
- Independencia
- Emular máquinas completas

Retos

- Consumo de recursos
- Tiempos de arranque
- Menos rendimiento que máquina real
- Pueden ocupar mucho espacio
- Cada máquina virtual requiere su propio sist. operativo invitado

Contenedores y docker

Virtualización a nivel de sistema operativo

Múltiples servidores aislados ejecutan en un servidor

El mismo kernel de sist. operativo implementa los servidores invitados

Requiere aislamiento de procesos completo a nivel de núcleo de sist. operativo

Docker (comienza en 2011) soporta contenedores

Varias partes

Especificación descripciones contenedores (imágenes)

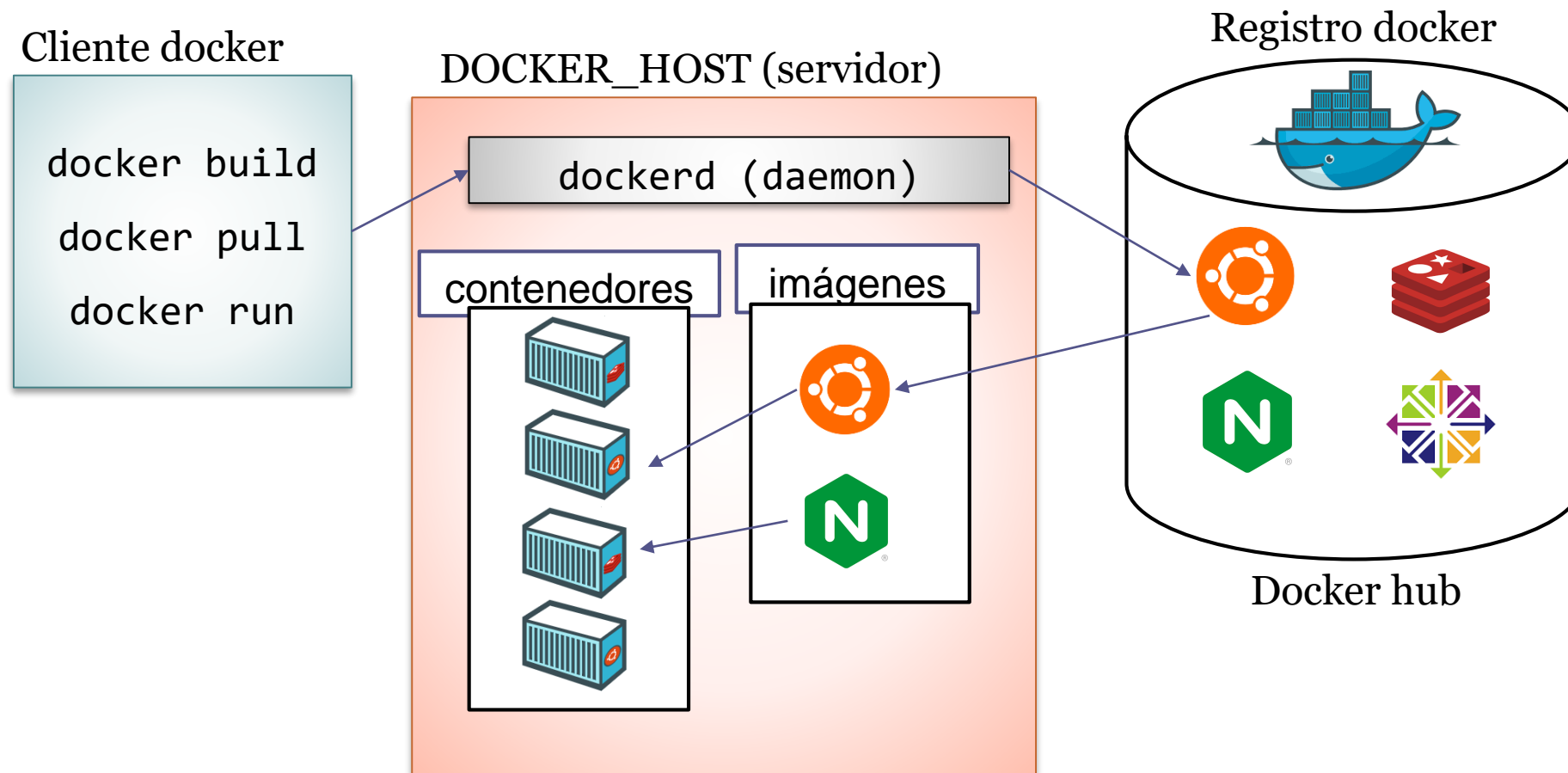
Plataforma ejecución contenedores

Registro contenedores (Docker-hub)



Arquitectura alto nivel docker

Arquitectura cliente-servidor



Imágenes Docker

Imagen contenedor = plantilla de solo lectura con instrucciones para crear un contenedor

Lenguaje dominio específico

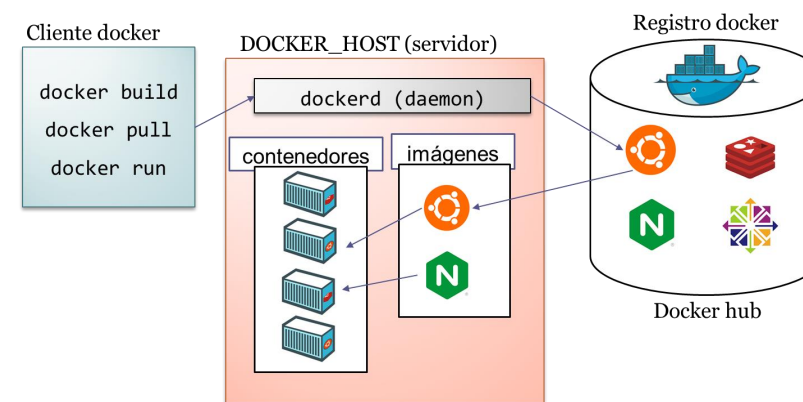
Habitualmente en un *Dockerfile*

Arquitectura en capas

Una imagen suele basarse en otra imagen + alguna personalización

Cada instrucción crea una capa en la imagen

Reutilización capas inferiores



Contenedores docker

Instancia de una imagen ejecutable

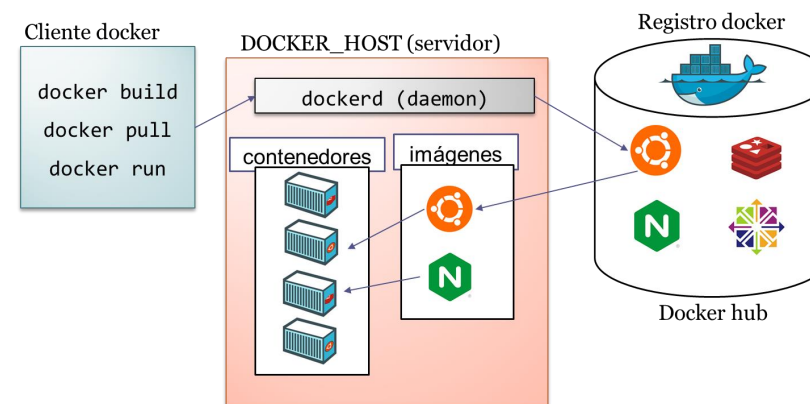
Los contenedores suelen estar aislados

De otros contenedores

De la máquina anfitrión

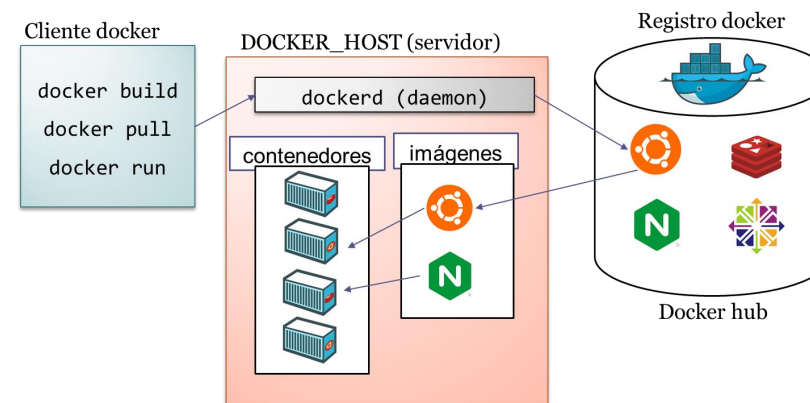
Es posible configurar el aislamiento

Volúmenes de datos, red, ...



Registro Docker

Base de datos de imágenes de contenedores
Docker Hub es un registro público (utilizado por defecto)
Es posible utilizar registros privados

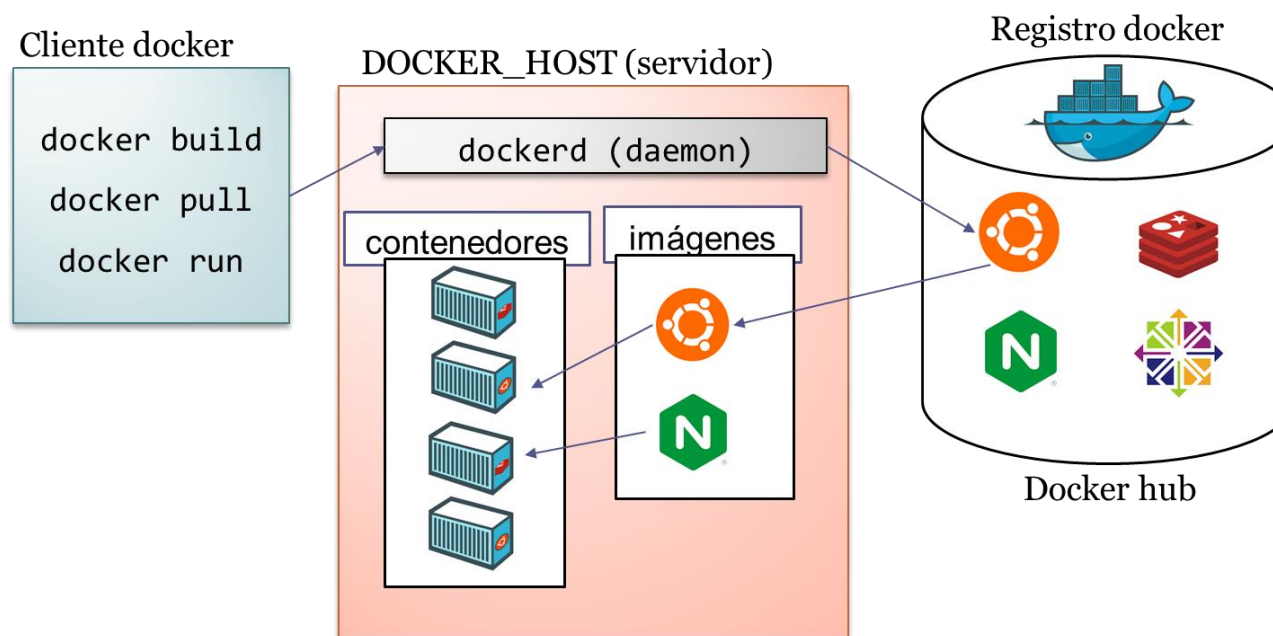


Ciente docker

Comando docker

Se comunica con el daemon docker utilizando el API

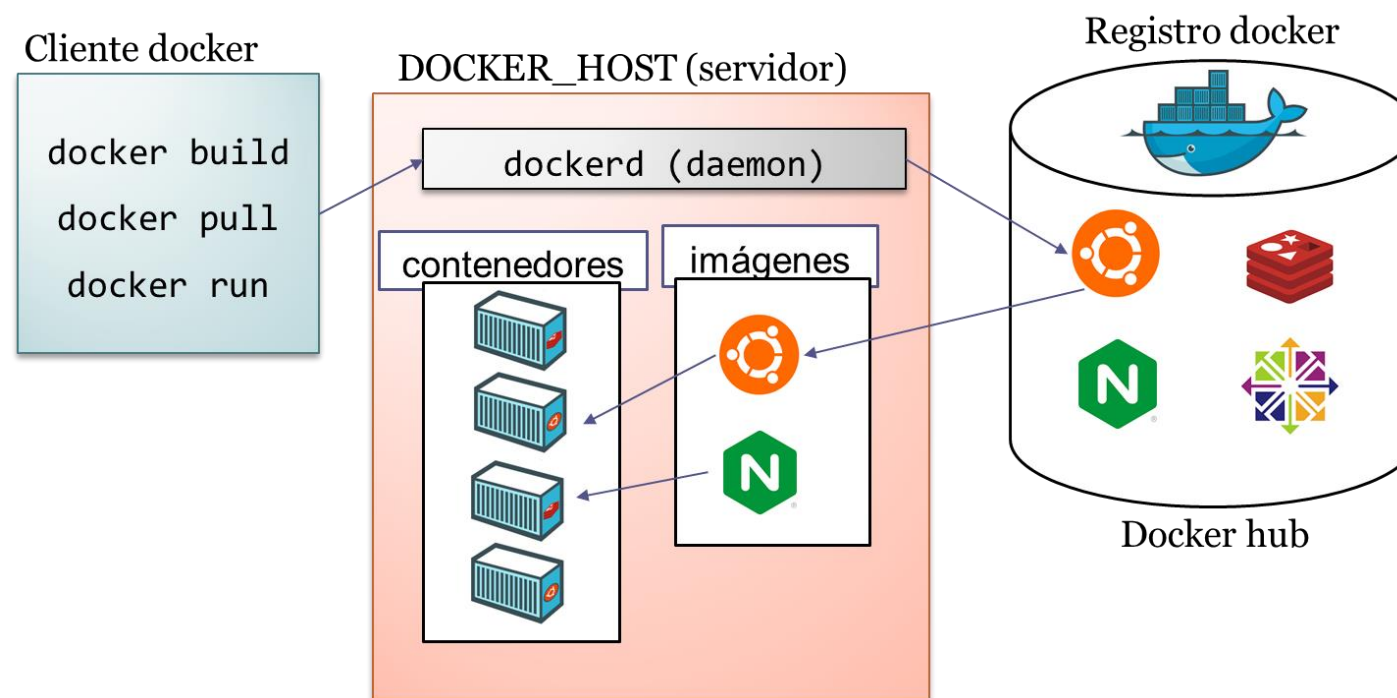
Comandos típicos: docker pull, docker run, ...



Daemon docker

El daemon docker (dockerd) escucha peticiones del API y gestiona imágenes y contenedores

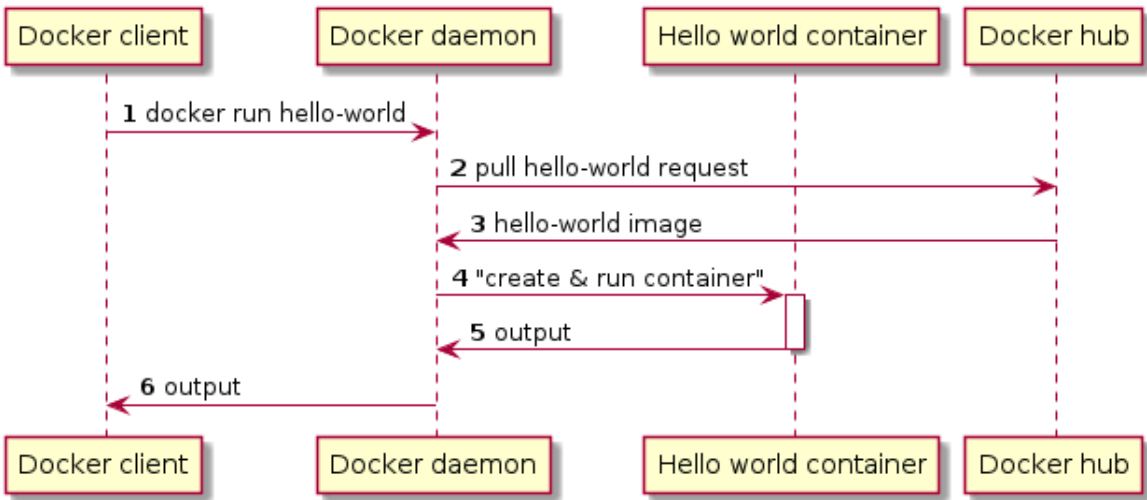
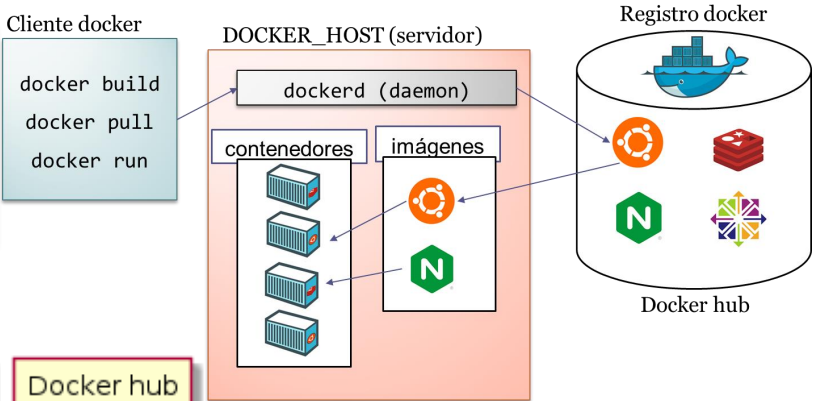
También puede comunicarse con otros demonios



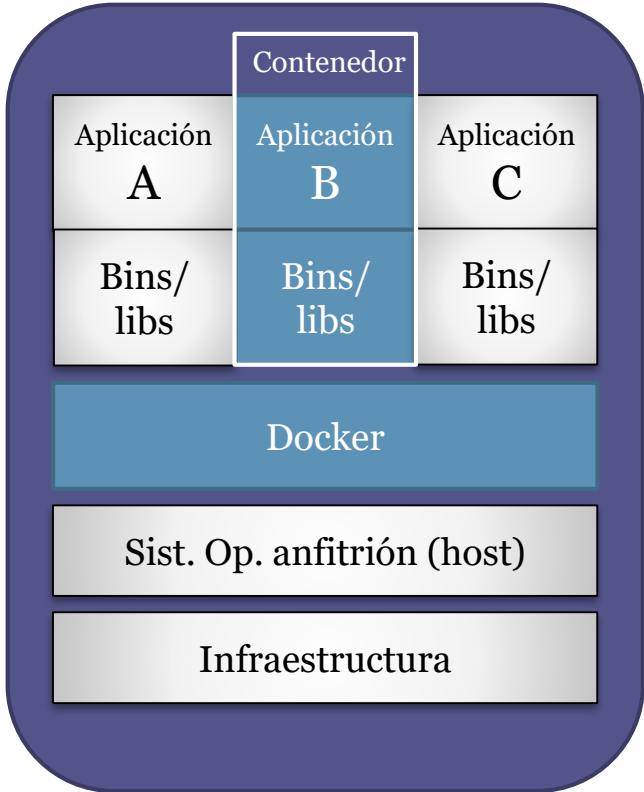
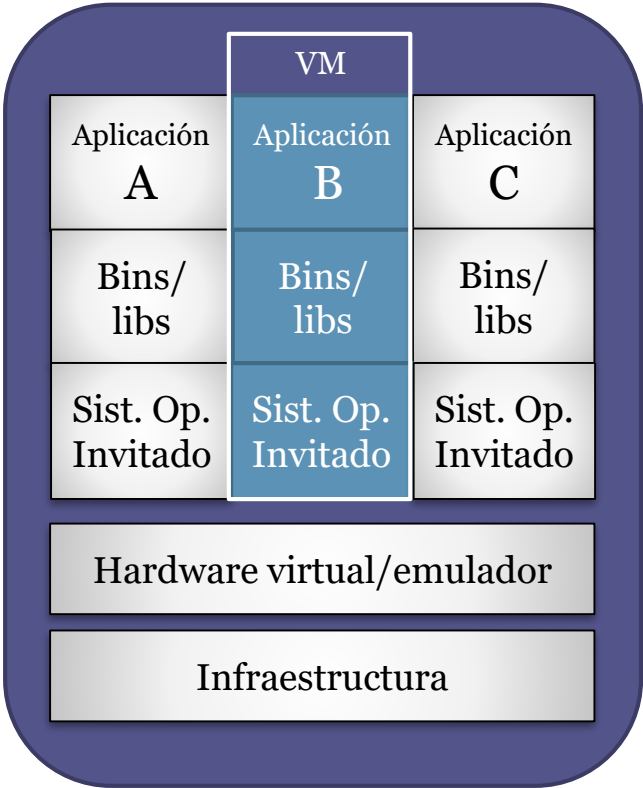
Ejemplo docker

Diagrama de secuencia para ejemplo hello-world

```
$ docker run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
1b930d010525: Pull complete
Digest: sha256:f9dfddf63636d84ef479d645ab5885156ae030f...
Status: Downloaded newer image for hello-world:latest
```



Máquinas virtuales vs Contenedores



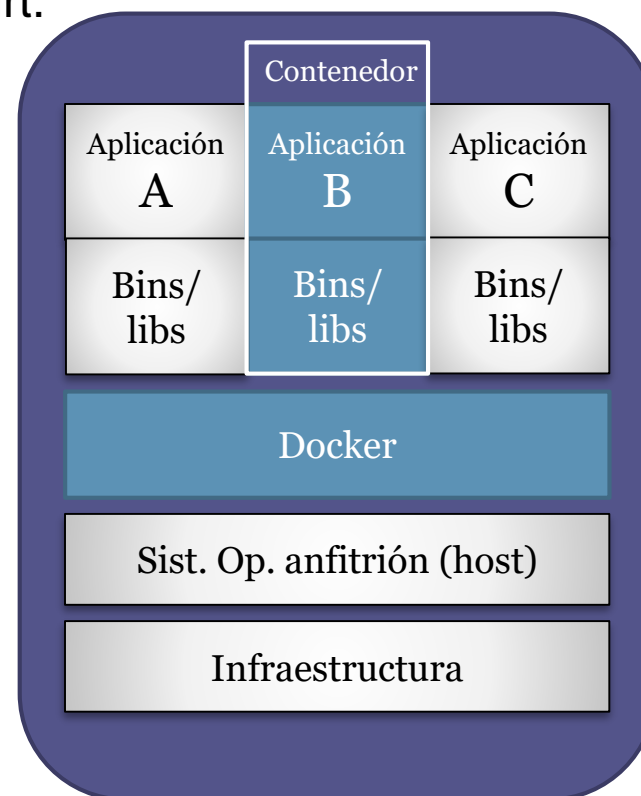
Consecuencias contenedores

Ventajas

Consistencia, portabilidad
 Fácil de desplegar
 Aislamiento
 Rendimiento
 Menos espacio que máquinas virt.
 1000s de contenedores
 Arquitectura inmutable
 Configuración declarativa
 Infraestructura como código
 Automatización

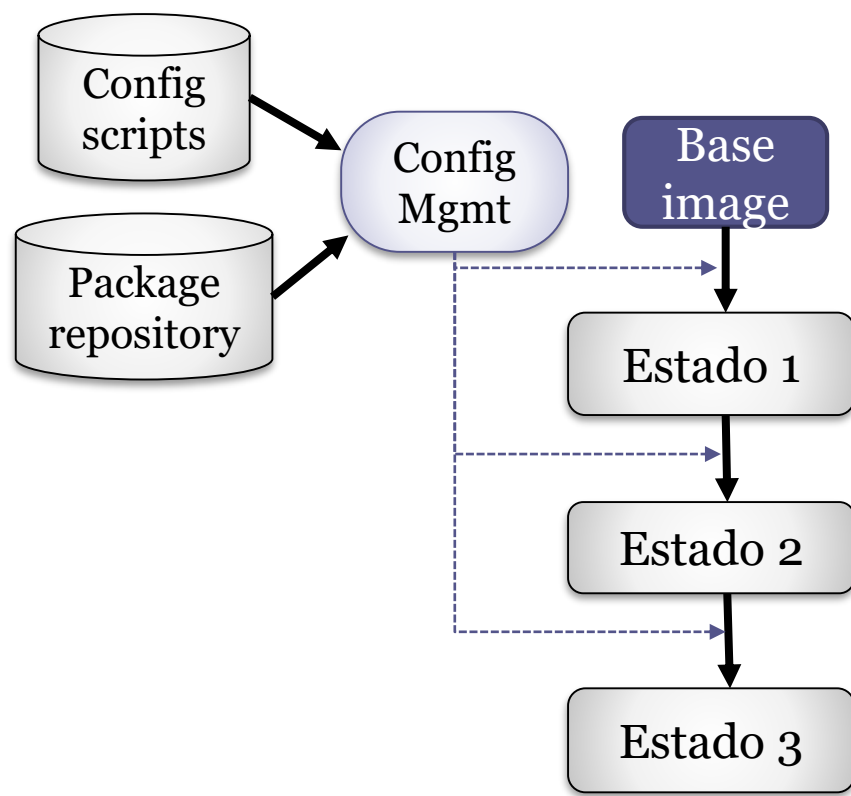
Retos

Orquestación
 Persistencia más compleja
 Aplicaciones gráficas
 Dependiente de plataforma (Linux)

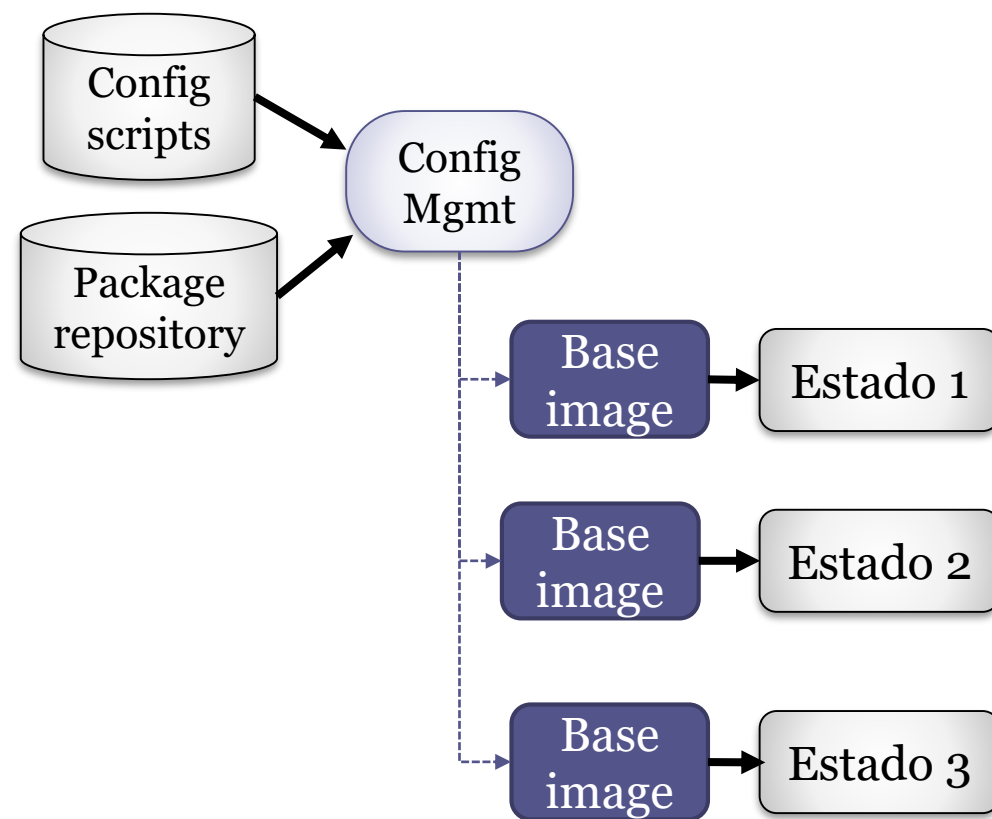


Infraestructura mutable vs immutable

Infraestructura mutable



Infraestructura inmutable



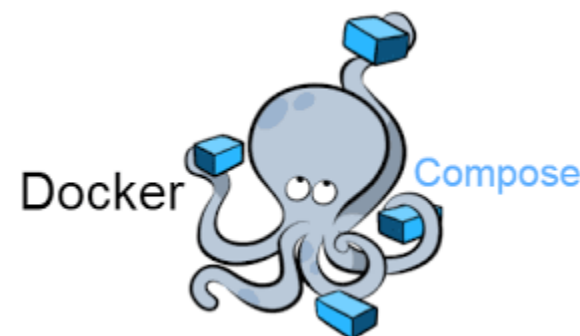
Gestor contenedores

Docker-compose = herramienta para definir y ejecutar múltiples aplicaciones contenedores

Fichero configuración YAML ([docker-compose.yml](#))

Con un commando simple, crea y arranca todos los servicios a partir de una configuración multi-contenedor

Docker-compose suele trabajar en un host sólo



Orquestación contenedores

Gestión automática de clusters de contenedores

Características típicas:

Balaneo de carga, ciclo vida contenedores, suministro,...

Kubernetes

Desarrollado inicialmente en Google, luego CNCF

Marco sistemas distribuidos

Los cluster consisten en pods, despliegues y servicios

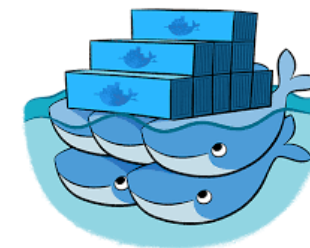
Disponible en la mayoría de proveedores de la nube



Docker swarm

Desarrollado por Docker

Puede considerarse un "modo" de ejecutar docker



Despliegue



Canal de despliegue

Implementación automatizada del proceso de construcción, despliegue, pruebas, y publicación

Objetivos

Crear entornos de ejecución bajo demanda

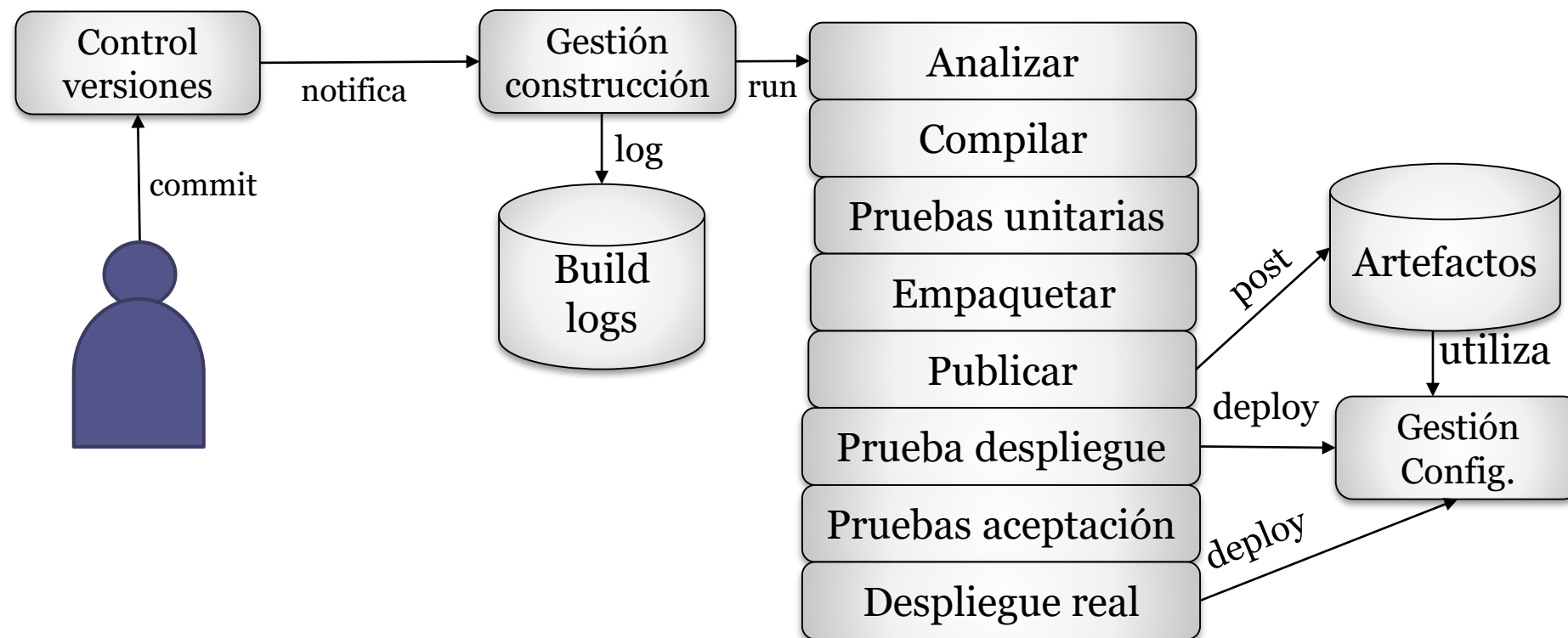
Resultados predecibles, rápidos, repetibles y fiables

Entornos de ensayo y producción consistentes

Bucles de realimentación rápidos para reaccionar

Hacer que los días de publicación de releases no tengan riesgo (que incluso sean aburridos)

Deployment pipeline



Despliegue manual

Círculo vicioso de tamaño y riesgo de despliegue



Despliegue continuo

"Si duele, hazlo más a menudo"

En el límite: "Hazlo continuamente"

Ejecutar el canal de despliegue en cada *commit*

Fase final: despliegue en producción

Posibilidades

- Confirmación por algún humano antes de ir a producción

- Despliegue automático en producción

- Despliegue en producción marcado por alguna etiqueta

Compromiso

- Coste de avanzar más rápido vs coste de errores en producción

Despliegue continuo

Patrones

Infraestructura como código

Mantener todo en control de versiones

Código

Configuración

Esquemas de datos

Documentación

Alinear desarrollo y operaciones (DevOps)

Herramientas:

Ansible, Chef, Puppet,...

Mejores prácticas: 12 factores (siguiente)

12 factores <https://12factor.net/>

I. Código base (Codebase): Un código base sobre el que hacer el control de versiones y múltiples despliegues

II. Dependencias: Declarar y aislar explícitamente las dependencias

III. Configuraciones: Guardar la configuración en el entorno

IV. Backing services: Tratar a los “backing services” como recursos conectables

V. Construir, desplegar, ejecutar: Separar completamente la etapa de construcción de la etapa de ejecución

VI. Procesos: Ejecutar la aplicación como uno o más procesos sin estado

VII. Asignación de puertos: Publicar servicios mediante asignación de puertos

VIII. Concurrencia: Escalar mediante modelo de procesos

IX. Desechabilidad: Hacer sistema más robusto intentando conseguir inicios rápidos y finalizaciones seguras

X. Paridad en desarrollo y producción:

Mantener desarrollo, preproducción y producción tan parecidos como sea posible

XI. Historiales: Tratar historiales como una transmisión de eventos

XII. Administración de procesos:

Ejecutar tareas gestión/administración como procesos que solo se ejecutan una vez

Software en producción



Atributos de calidad en producción

Configurabilidad

Personalizar el sistema sin recompilarlo

Observabilidad

Monitorizar estado interno del sistema

Disponibilidad

Provabilidad que el sistema funcione en un tiempo t

Estabilidad

Producir disponibilidad a pesar de faltas y errores

Fiabilidad

Probabilidad que un sistema produzca salidas correctas después de algún tiempo t

Configurabilidad

Muchas propiedades configurables

Hostnames, núms. puerto, localizaciones en sistema ficheros, núms. ID, nombres usuario, passwords, etc.

Ficheros configuración = interfaz entre desarrolladores y operaciones

Deberían ser legibles por humanos y procesables automáticamente

Ejemplos: XML, JSON, YAML, ...

Pueden contener información sensible

Separados de código fuente



Logging

Logging es universal y fácil de generar

Tecnología de caja blanca (integrado en código fuente)

Muestran actividad y pueden persistir fácilmente

Legibles por humanos

Localizaciones de los log

Separar logs de código fuente

Niveles de Logging

Encontrar un buen balance entre demasiado ruido y silencio

Cualquier cosa marcada como "ERROR" ó "SEVERE" debería requerir acción

Importante: deshabilitar logs de depuración en producción



Monitorización

Monitorizar: Observar el comportamiento en tiempo de ejecución mientras el Sistema funciona

Sistemas base datos series temporales

Visualizaciones series y cuadros de mandos

Prometheus, Graphite, Grafana, Datadog, Nagios, ...

Chequeos de salud

Profiling: Medir rendimiento de un software mientras se ejecuta



Datos en producción

Replicación de datos y alta disponibilidad

Asegurar backup y restauración

Esquemas base datos en control de versiones

Gestión de peticiones de cambios

Migración de datos

Purga de datos (data purging)

Datos sensibles en producción

Inaccesibles a desarrolladores

Encriptados

. . .



Problemas en sistemas

Falta (fault):

Estado interno incorrecto, iniciado por un defecto o inyectado

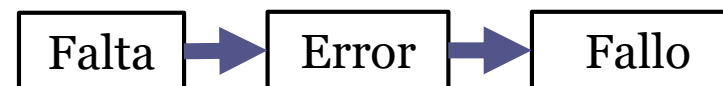
Error:

Operación incorrecta observable

Fallo (failure):

Falta de disponibilidad. Sistema que no responde

Reacciones en cadena



Ley de los sistemas grandes

Los sistemas grandes existen en un estado de fallo parcial continuo

Corolario:

"Todo está funcionando" es una anomalía

Importante:

Evitar propagación de faltas



Source: "Airplane" film

<https://www.imdb.com/title/tt0080339/>

Patrones en producción

Load balancing

Timeouts

Circuit breakers

Bulkheads

Steady state

Fail fast

Let it crash

Handshaking

Test harnesses

Decoupling middleware

Create backpressure

Governor



Some libraries: <https://resilience4j.readme.io/>

Load balancing

Distribuir peticiones entre varias instancias

Objetivo: server todas las peticiones correctamente en el tiempo más corto posible

Decisiones a tomar:

- Algoritmos de balanceo de carga

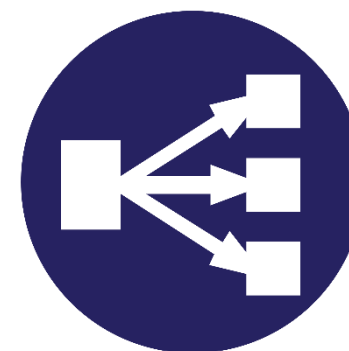
- Qué chequeos de salud realizar en las instancias

- Qué hacer cuando ninguna instancia está disponible

- Sincronización entre instancias

Balanceadores de carga

- Hardware/Software



Timeouts

Añadir un limitador temporal a peticiones

Proporciona aislamiento de faltas

Un problema en algún otro servicio no debería ser un problema nuestro

Timeouts normalmente van seguidos de reintentos

Los reintentos podrían empeorar las cosas

La situación no suele recuperarse automáticamente

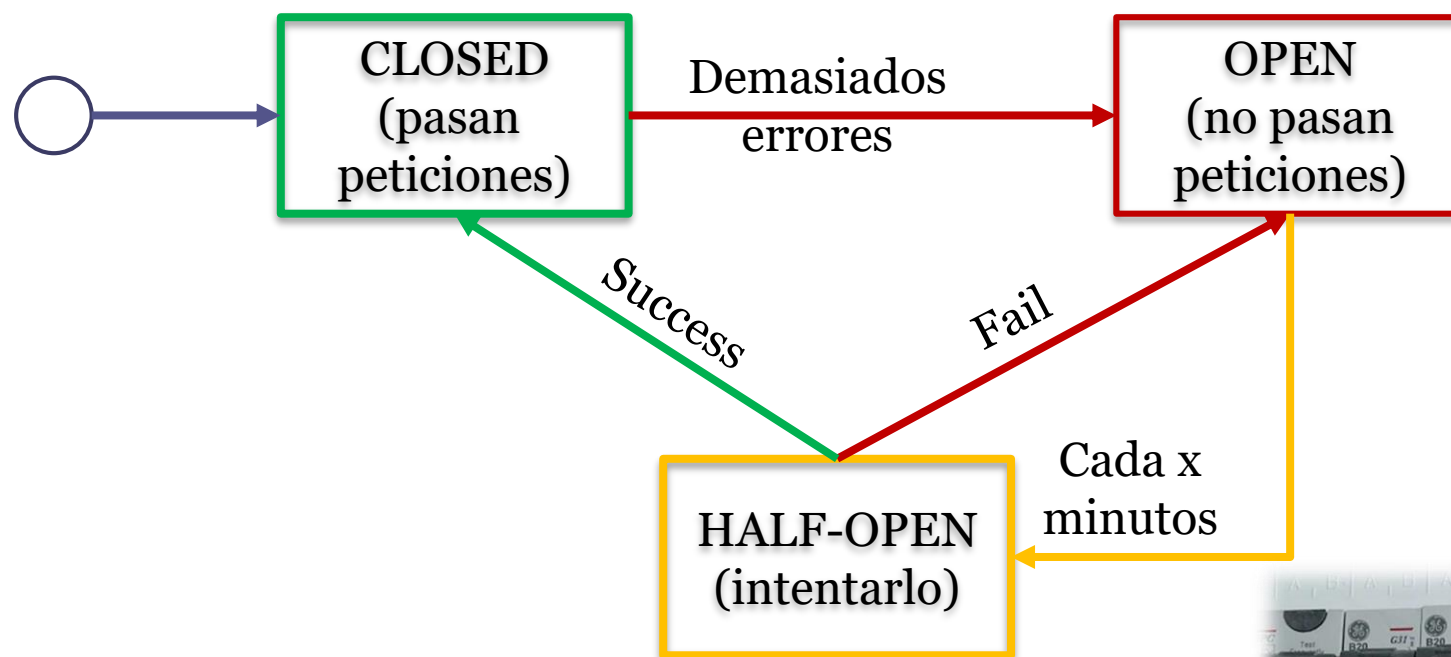
El consumidor espera más tiempo

A veces, mejor fallar rápido



Circuit breaker

Inspirado en fusibles eléctricos

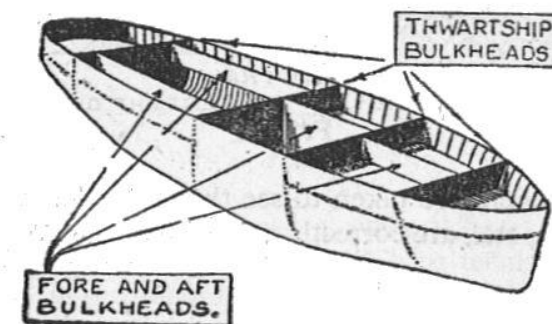
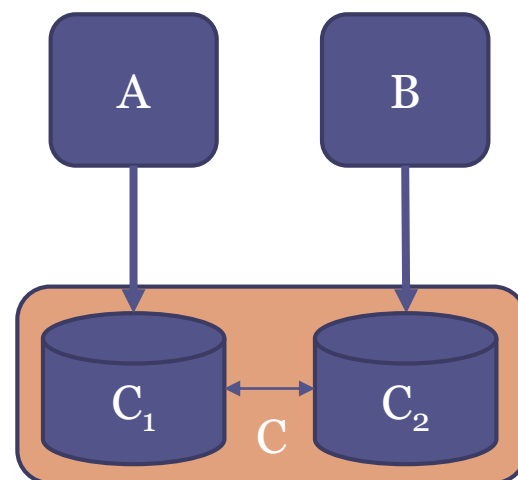
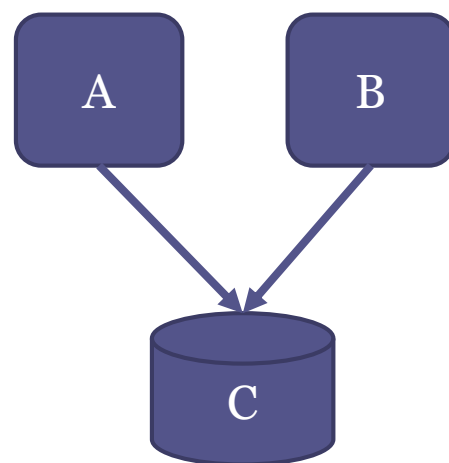


Bulkheads

"Contener el daño" (salvar parte del barco)

Si un componente se rompe, el sistema puede funcionar

Ejemplo: replicar instancias en la nube



Estado estacionario (*steady state*)

Mantener los recursos del sistema constantes

Evitar intervención humana para limpieza

Ejemplos:

- Purgar datos

- Gestión de ficheros de Log

- Memoria caché



Fail fast

No hacer esperar a los consumidores por respuestas de fallo

Reservar recursos antes de empezar el trabajo

No realizar trabajo inútil

Verificar los puntos de integración al principio

Chequear que los recursos están disponibles

Validación básica de entrada

Distribuir la carga

Rechazar peticiones nuevas cuando la carga es muy elevada



"Chequear ingredientes antes de cocinar"

Let it crash

"Romper components para salvar el sistema"

Inspirado por gestión de errores de Erlang

Si un componente no puede hacer lo que tiene que hacer, dejarlo que rompa

Dejar que otro componente haga la recuperación

No programar defensivamente

Condiciones

Crear límites

Un componente rompe aisladamente

Sustitución rápida

Supervisión

Reintegración



Handshaking

"Acordar antes de hacer algo"

Control de demanda cooperativo

Cliente y servidor acuerdan hacer algo

El servidor puede rechazar trabajo nuevo

Los servicios proporcionan una consulta de chequeo de salud (health check)

Balanceadores de carga chequean la salud de las instancias antes de la petición



Crear *backpressure*

Backpressure = resistencia que se opone al flujo de datos

Cuando la entrada viene más rápido que la salida

Crear seguridad ralentizando a los productores

Estrategias

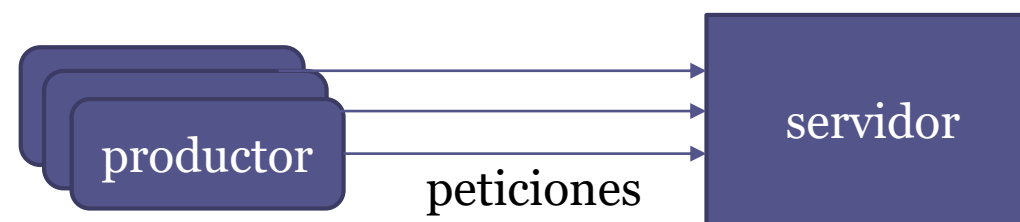
Controlar/ralentizar productores

Buffer (acumular datos de entrada temporalmente)

Buffers sin límites pueden ser peligrosos

Ignorar peticiones

No siempre es aceptable perder datos



Regulador (*governor*)

Crear reguladores que ralentizan la velocidad de algunas acciones

Cuando algo automatizado se hace mal, se puede hacer mal de forma muy rápida

Evitar multiplicadores de fuerza

Ralentizar las cosas para permitir intervención humana

Aplicar resistencia en acciones poco seguras

Ejemplos: borrar instancias, apagado de máquinas,...

Considerar una curva de respuesta



Source: [Steam Engine Centrifugal Governor](#)

Test harnesses

Chequean la mayoría de los modos de fallo en las pruebas

Emular fallos fuera de la especificación

Poner en tensión al consumidor

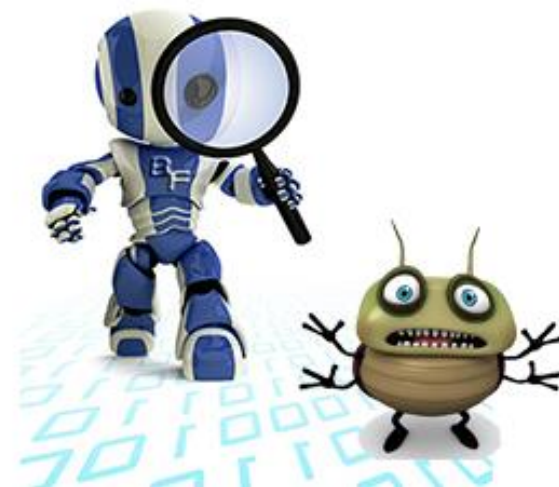
Producir respuestas lentas, no respuestas, respuestas basura, ...

Estos sistemas de prueba pueden compartirse

Ejemplo: *servicios asesinos*

Relacionado con Ingeniería del caos

[Ver siguiente]



Ingeniería del caos

Propuesto por Netflix en 2010 (*Chaos Monkey*)

Introducir fallos adrede en los sistemas

Probar sistemas distribuidos

Romper cosas a propósito

Pruebas mediante inyección de fallos

Asegurar que el fallo de una instancia no afecta al sistema global

Anti-fragilidad y resiliencia

Capacidad para absorber perturbaciones



<https://github.com/Netflix/chaosmonkey>

Antipatrones en producción

- Integration points
- Chain reactions
- Cascading failures
- Users
- Blocked threads
- Self-denial attacks
- Scaling effects
- Unbalanced capacities
- Dogpile
- Force multiplier
- Slow responses
- Unbounded result sets



Pruebas en producción

Despliegue progresivo

Reducir radio de explosión de nuevos despliegues

Permitir experimentación

Algunas técnicas

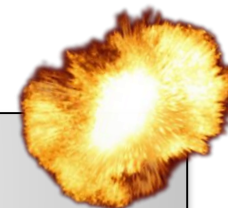
Canary releases

Feature toggles

Pruebas A/B y multi-armed bandits

Radio de explosión de un despliegue:

¿Quién es impactado? ¿Qué funcionalidad? ¿Cuántos sitios?

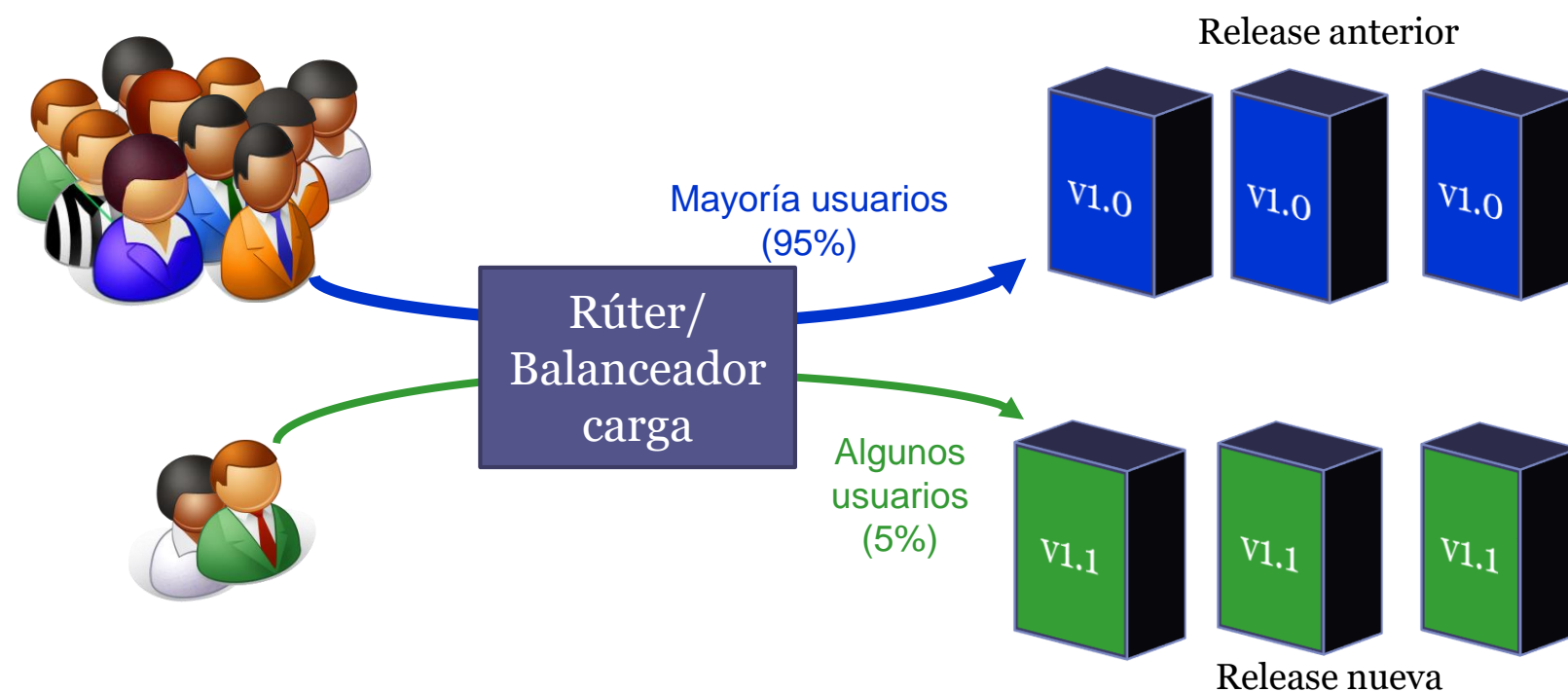


Canary release

Introducir nuevas releases ofreciendo lentamente el cambio a un subconjunto de usuarios

Desde infraestructura (rúter/balanceador de carga)

Despliegue Blue-Green



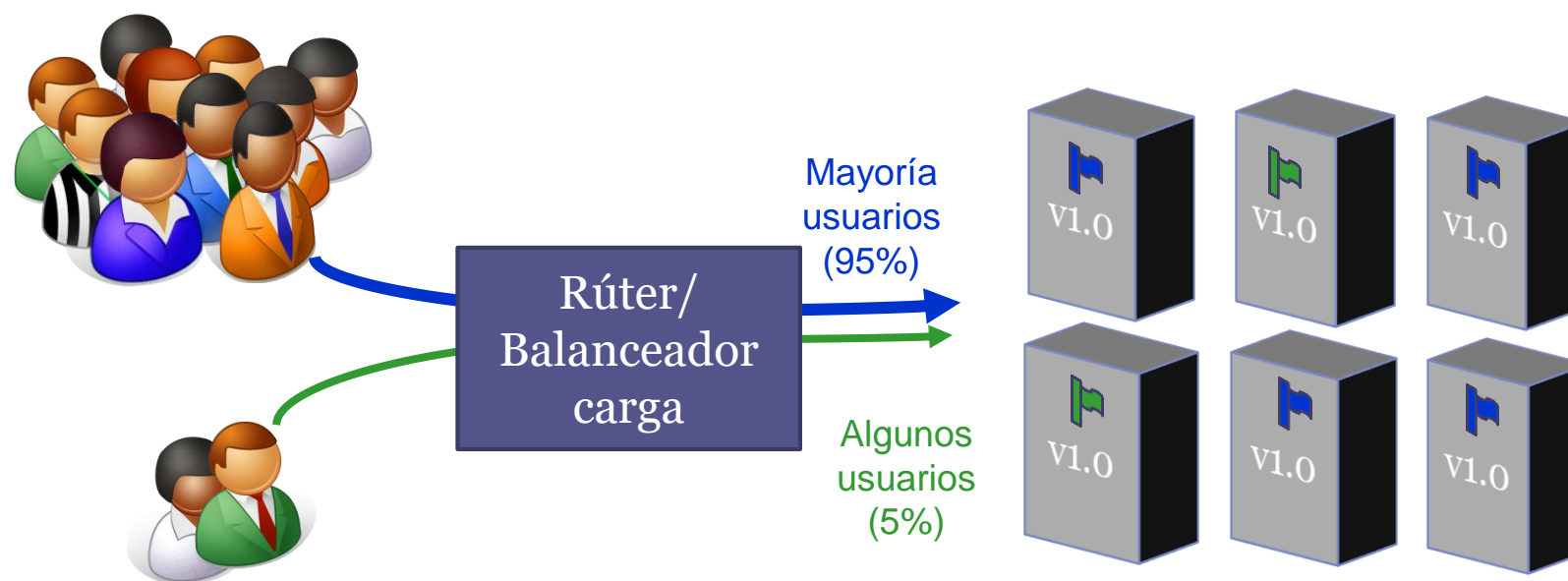
<https://martinfowler.com/bliki/CanaryRelease.html>

Feature toggles

También conocidos como *feature flags*, *feature bits*

Modificar comportamiento Sistema sin cambiar código

Desacoplar despliegue de *release*



<https://martinfowler.com/articles/feature-toggles.html>

Tipos de pruebas en producción

Pruebas A/B:

También conocidas como split testing, bucket testing

Experimento controlado para probar alguna hipótesis

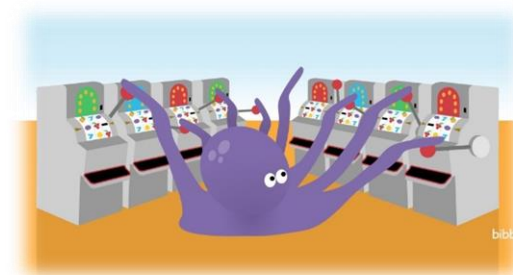
Dividir usuarios en grupos

Problema: malas alternativas son mostrados a grupos de usuarios durante experimento

Multi-armed bandits

Disposición dinámica del tráfico

Intentar que las alternativas malas tengan menos usuarios



Pruebas de carga y estrés

Pruebas de carga

Medir rendimiento bajo una carga

Ejemplo: similar múltiples usuarios accediendo a la vez

Pruebas de estrés

Carga llevado más allá de patrones normales para probar respuesta del Sistema

Probar límites superiores

¿Qué ocurre cuando se alcanza límite?

Varias herramientas

JMeter, Gatling



Incidentes y post-mortem

Resolver y revisar un incidente

Asegurar que el equipo lo ve sin buscar culpables

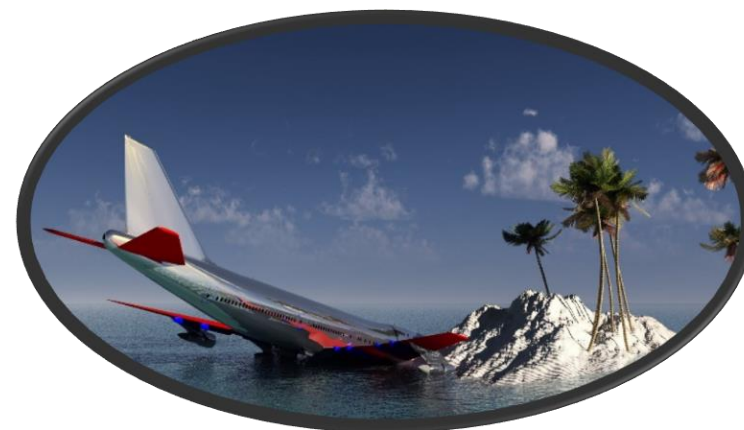
Crear informe *post-mortem*

- Detalles del incidente

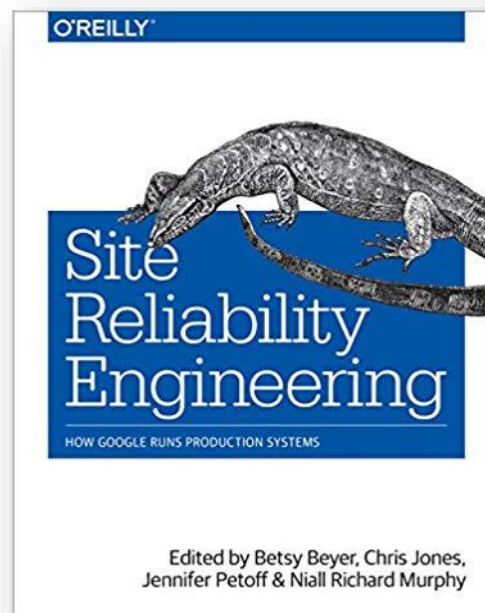
- Línea temporal y acciones tomadas para resolverlo

- Análisis de causa raíz (*root cause*)

Identificar medidas preventivas



Fin de la presentación



Free online