Universidad de Oviedo

ES
Español

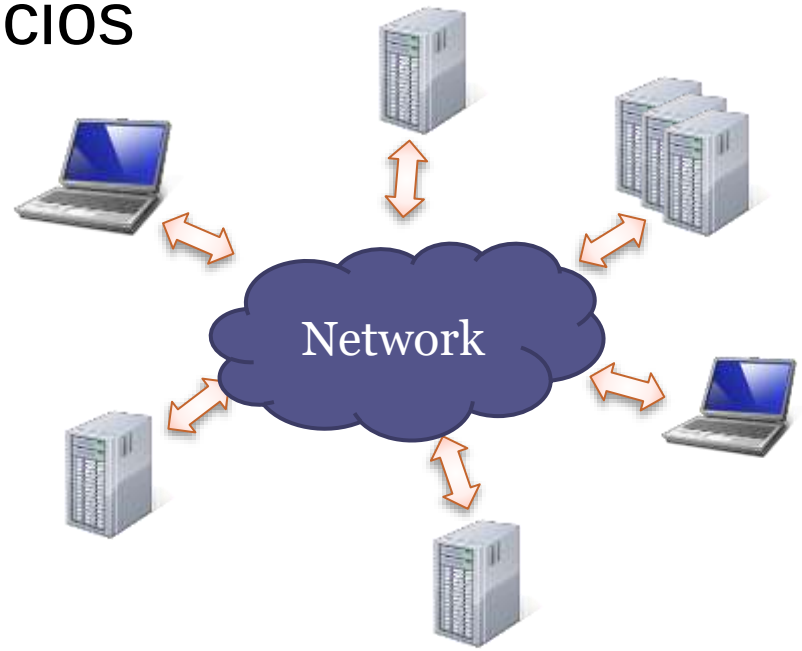
Sistemas distribuidos
Sistemas escalables y Big data

2023-24

Jose Emilio Labra Gayo

Sistemas distribuidos

- Estilos de integración
- Topologías: Hub & Spoke, Bus
- Patrón Broker
- Peer-to-peer
- Arquitecturas orientadas a servicios
  - WS-*, REST
- Microservicios
- Serverless



Estilos de integración

Transferencia de ficheros

Base de datos compartida

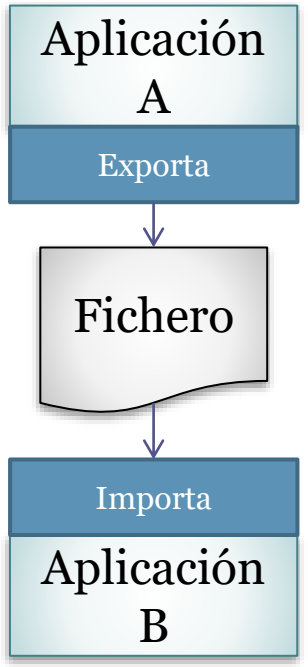
Invocación procedimiento remoto

Mensajería

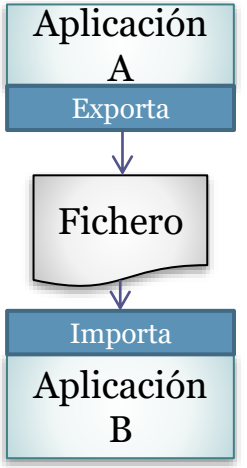
Transferencia de ficheros

Una aplicación genera un fichero de datos que es consumido por otra

Una de las soluciones más comunes



Transferencia de ficheros



Ventajas

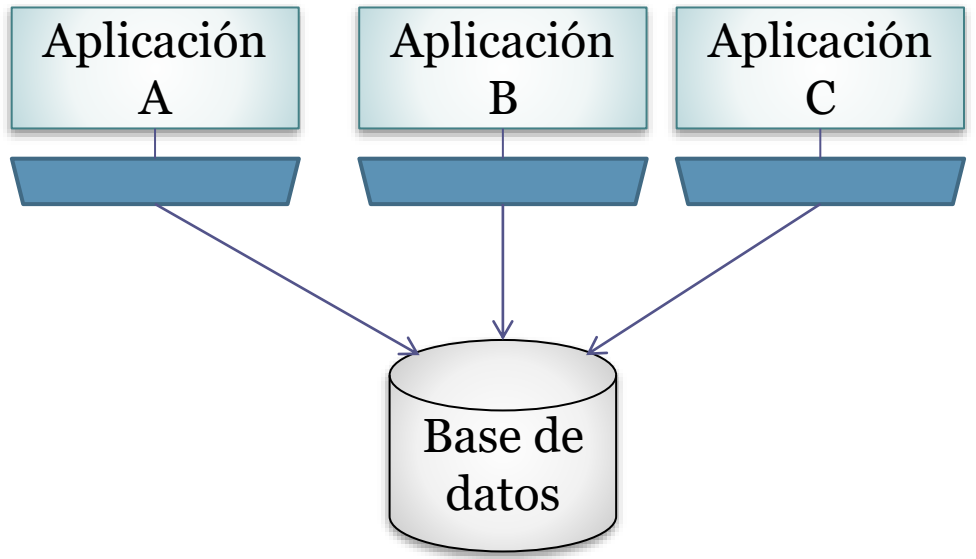
- Bajo acoplamiento
  - Independencia entre aplicación A y B
- Facilita depuración
  - Se pueden analizar datos del fichero

Problemas

- Acordar formato de fichero común
  - Puede aumentar acoplamiento
- Coordinación
  - Una vez enviado el fichero, la aplicación B puede modificarlo ⇒ ¡2 ficheros!
- Suele requerir intervención manual

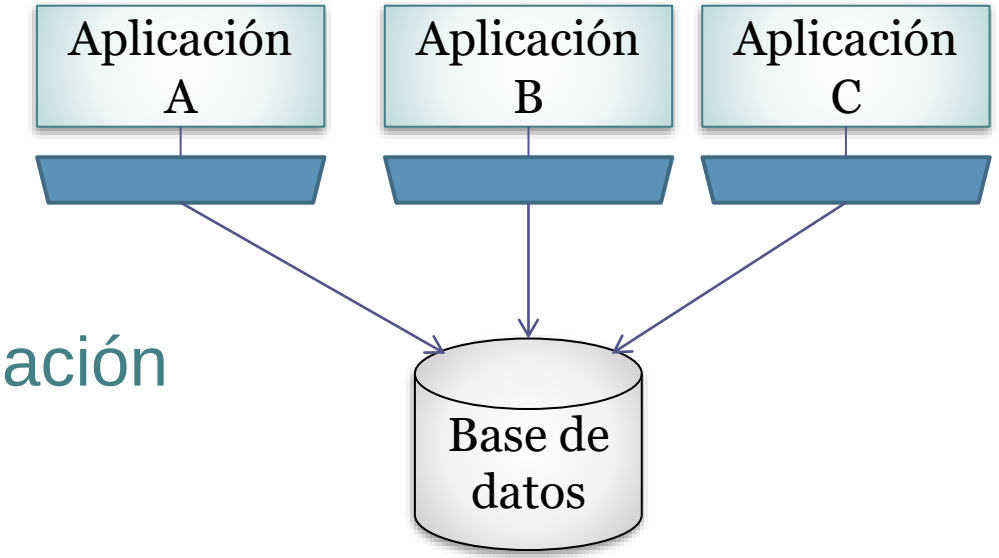
Base de datos compartida

Las aplicaciones almacenan sus datos en una base de datos común

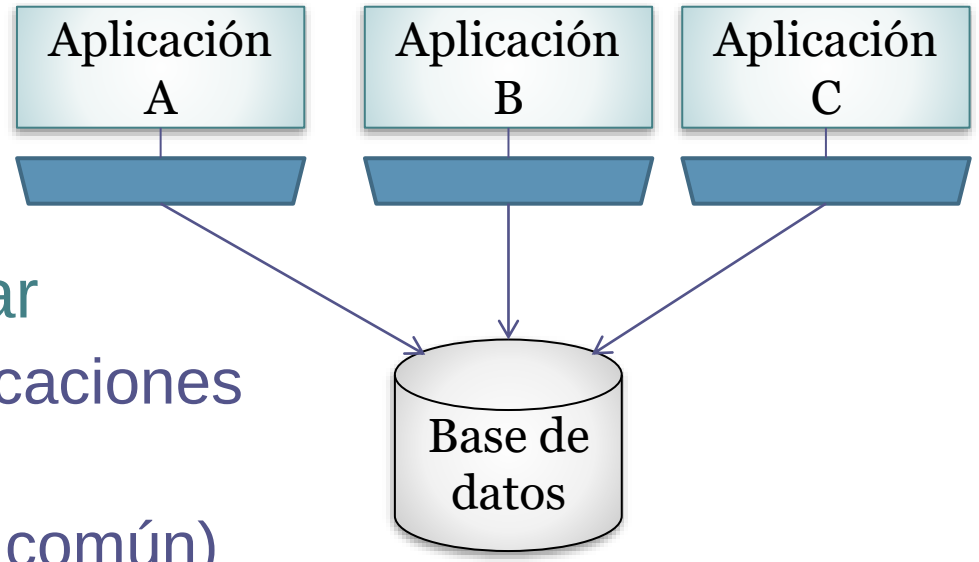


Base de datos compartida

- Ventajas
  - Datos siempre disponibles
  - Todo el mundo accede a la misma información
  - Consistencia
  - Formato familiar
    - SQL para todo?



Base de datos compartida



Problemas

- El esquema de la base de datos puede variar
  - Requiere esquema común para todas las aplicaciones
    - Fuente de problemas y conflictos
  - Necesidad de paquetes externos (acceso BD común)
- Rendimiento y escalabilidad
  - Base de datos como cuello de botella
- Sincronización
  - Problema con bases de datos distribuidas
  - Escalabilidad
  - NoSQL ?

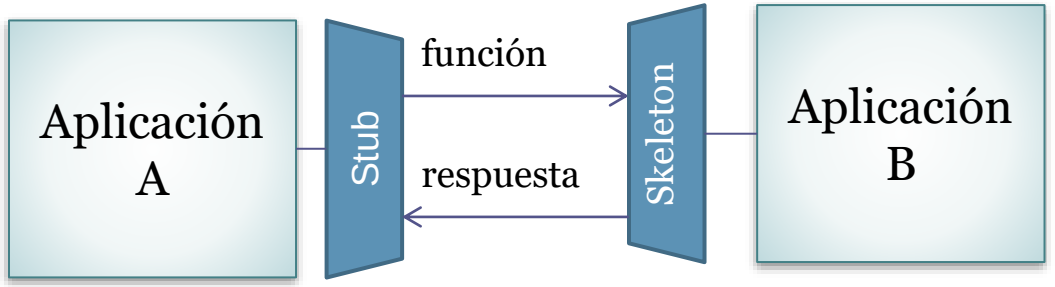
Base de datos compartida

Variaciones

- *Data warehousing*: Base de datos utilizada para análisis de datos e informes
- ETL: proceso basado en tres fases
  - Extracción: Obtención de fuentes heterogéneas
  - Transformación: Procesado de los datos
  - Carga (Load): Almacenamiento en base de datos compartida

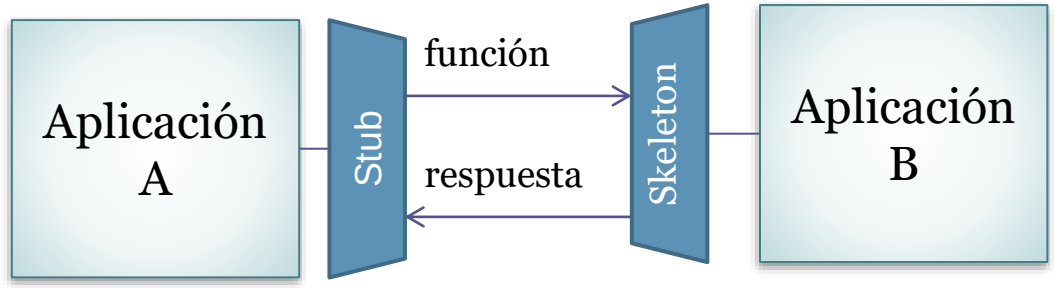
Invocación procedimiento remoto

- Una aplicación invoca una función de otra aplicación que puede estar en otra máquina
  - En la invocación puede pasar parámetros
  - Obtiene una respuesta
- Gran variedad de aplicaciones
  - RPC, RMI, CORBA, .Net Remoting, ...
  - Servicios web, ...



Invocación procedimiento remoto

- Ventajas
  - Encapsulación de implementación
  - Múltiples interfaces para la misma información
    - Se pueden ofrecer distintas representaciones
  - Modelo familiar para desarrolladores
    - Similar a llamar a un método



Invocación procedimiento remoto

Problemas

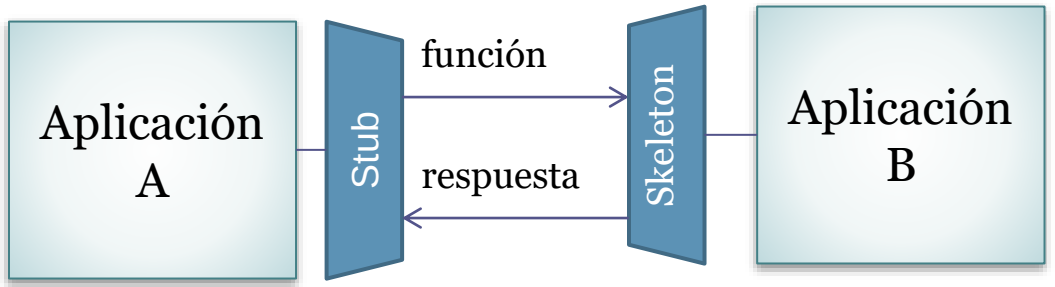
Falsa sensación de sencillez

Procedimiento remoto ≠ Procedimiento

8 falacias de computación distribuida

Invocaciones mediante sincronización

Aumenta acoplamiento entre aplicaciones



La red es fiable
La latencia es cero
El ancho de banda es infinito
La red es segura
La topología no cambia
Hay un administrador
El coste de transporte es cero
La red es homogénea

8 falacias computación distribuida

Invocación procedimiento remoto

Nuevas revisiones: gRPC

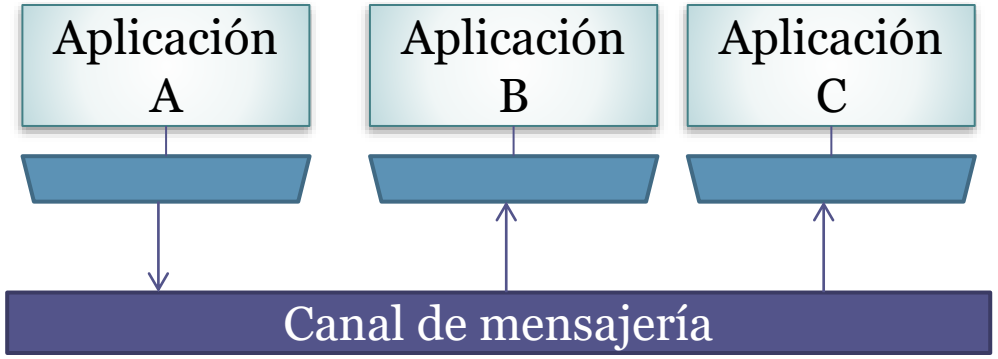
Propuesta de Google (https://grpc.io/)

Marco de aplicaciones de mensajería de alto rendimiento

Basado en http/2

Mensajería

- Múltiples aplicaciones independientes se comunican enviando mensajes a un canal
- Comunicación asíncrona
  - Las aplicaciones envían mensajes y continúan ejecutándose



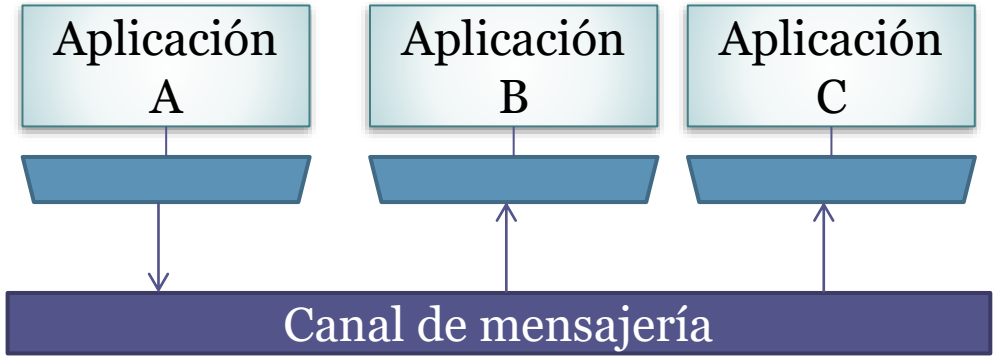
Mensajería

Ventajas

- Bajo acoplamiento
  - Aplicaciones independientes entre sí
- Comunicación asíncrona
  - Las aplicaciones continúan la ejecución
- Encapsulación
  - Sólo se expone el tipo de mensajes

Problemas

- Complejidad de implementación
  - Comunicación asíncrona
- Transformación de datos
  - Adaptación formato de mensajes
- Diferentes topologías



Topologías de integración

Hub & Spoke

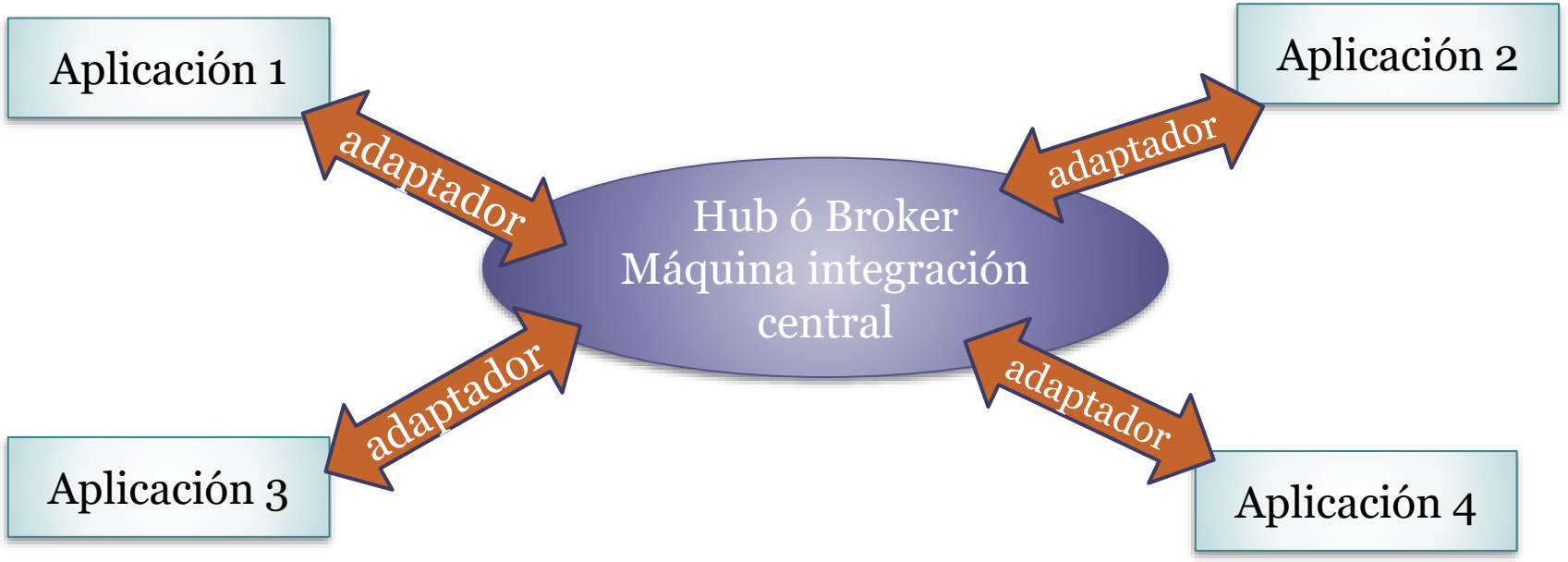
Bus

Hub & Spoke (central/radial)

Relacionado con patrón Bróker

Hub = Bróker centralizado de mensajes

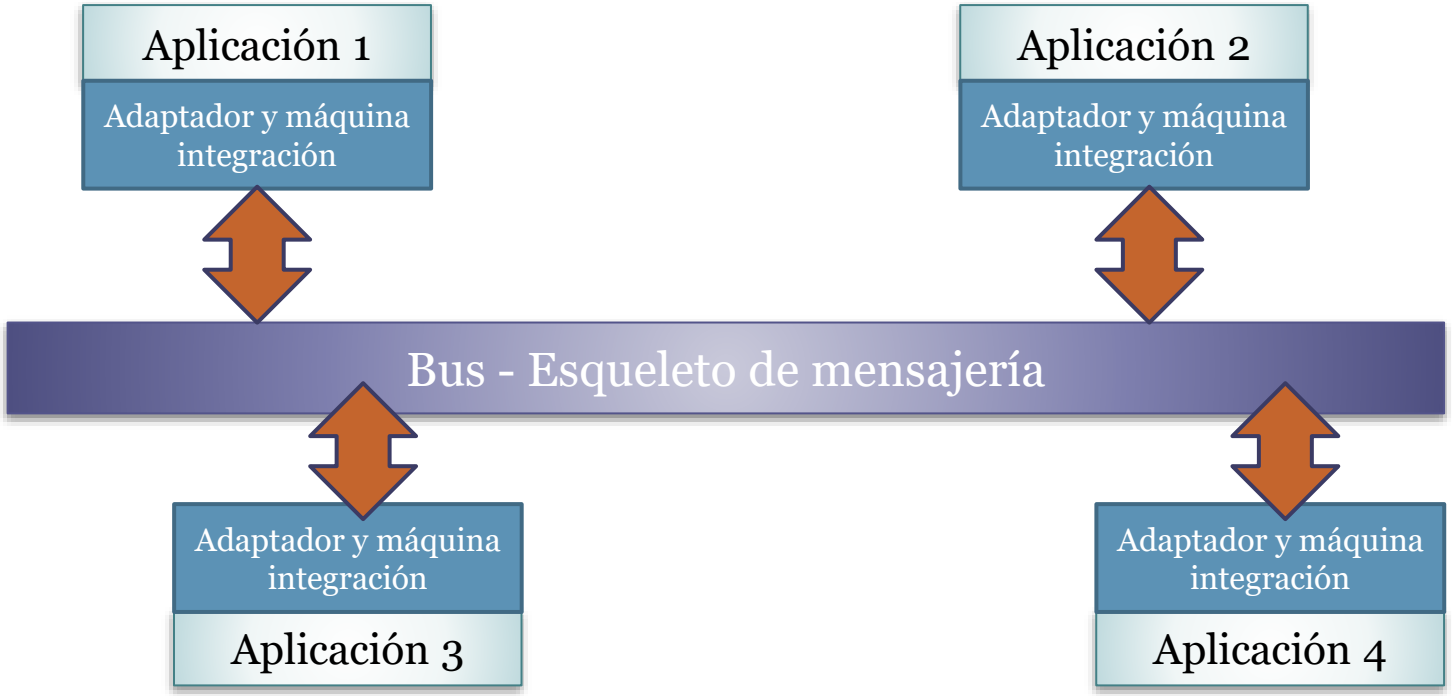
Se encarga de la integración



Bus

Cada aplicación contiene su máquina de integración

Estilo Publish/Subscribe

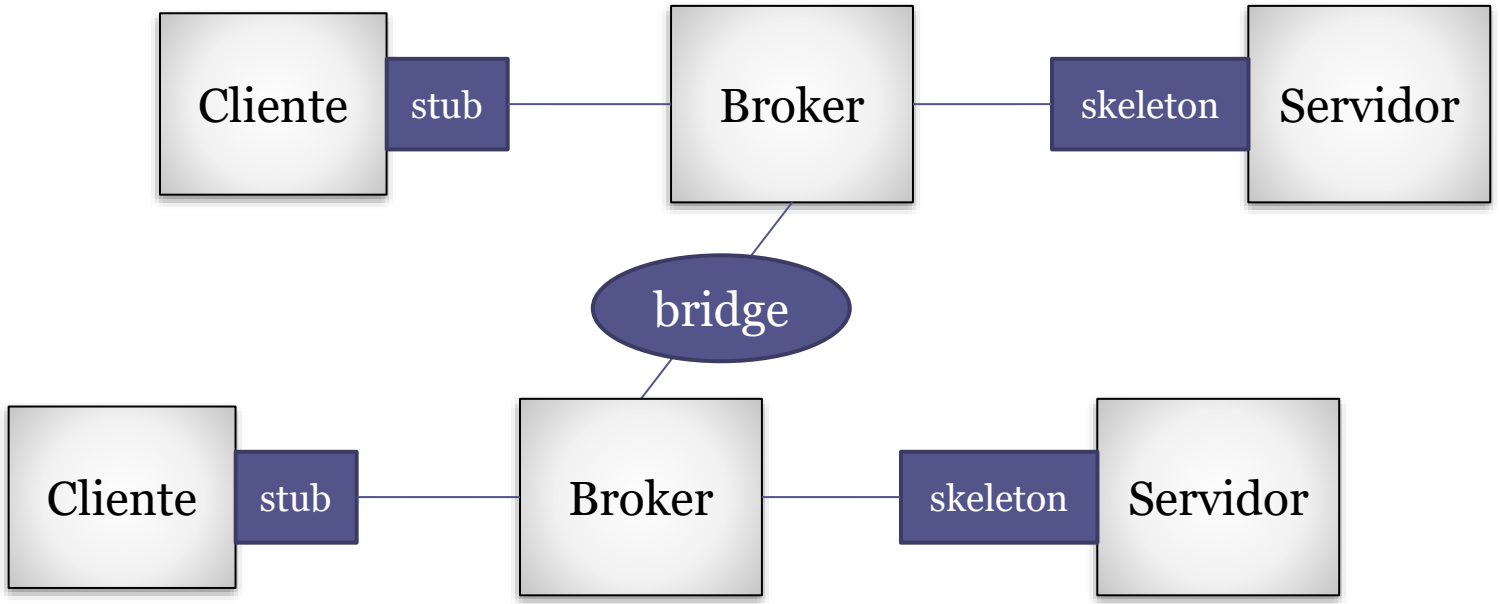


Bus

- ESB - Enterprise Service Bus
- Define un esqueleto (*backbone*) de mensajería
  - Conversión de protocolos
  - Transformación de formatos
  - Enrutamiento
- Proporciona un API para desarrollar servicios
  - MOM (Message Oriented Middleware)

Patrón Bróker

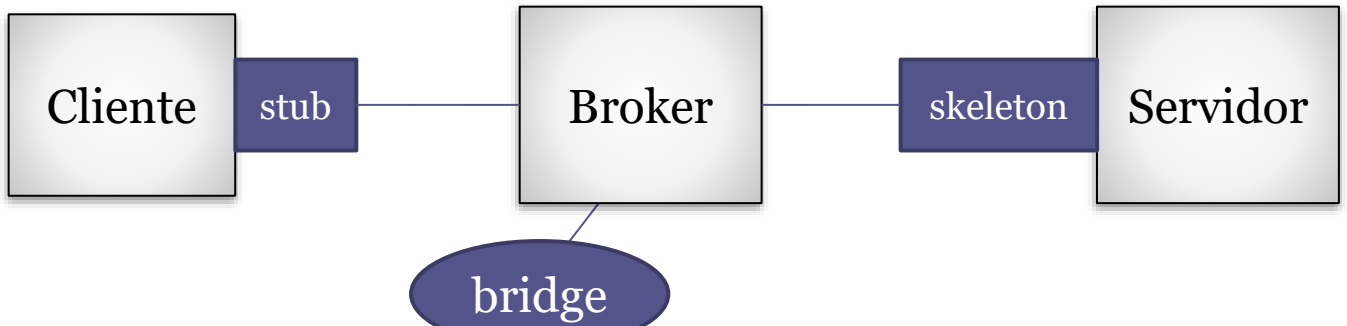
Nodo intermediario que gestiona la comunicación entre un cliente y un servidor



Patrón Bróker

Elementos

- Bróker
  - Se encarga de la comunicación
- Cliente: Envía peticiones
  - Proxy de cliente: *stub*
- Servidor: Devuelve respuestas
  - Proxy de servidor: *skeleton*
- Bridge: Puede conectar brókers entre sí



Patrón Bróker

Ventajas

- Separación de responsabilidades
  - Delega aspectos comunicación al bróker
  - Mantenimiento por separado
  - Reutilización
- Servidores independientes de clientes
- Portabilidad
  - Bróker = aspectos de bajo nivel
- Interoperabilidad
  - Mediante *bridges*

Problemas

- Rendimiento
  - Se añade una capa de indirección
  - Comunicación directa siempre va a ser más rápida
- Puede suponer acoplamiento fuerte entre componentes
- Bróker = punto de fallo único en el sistema

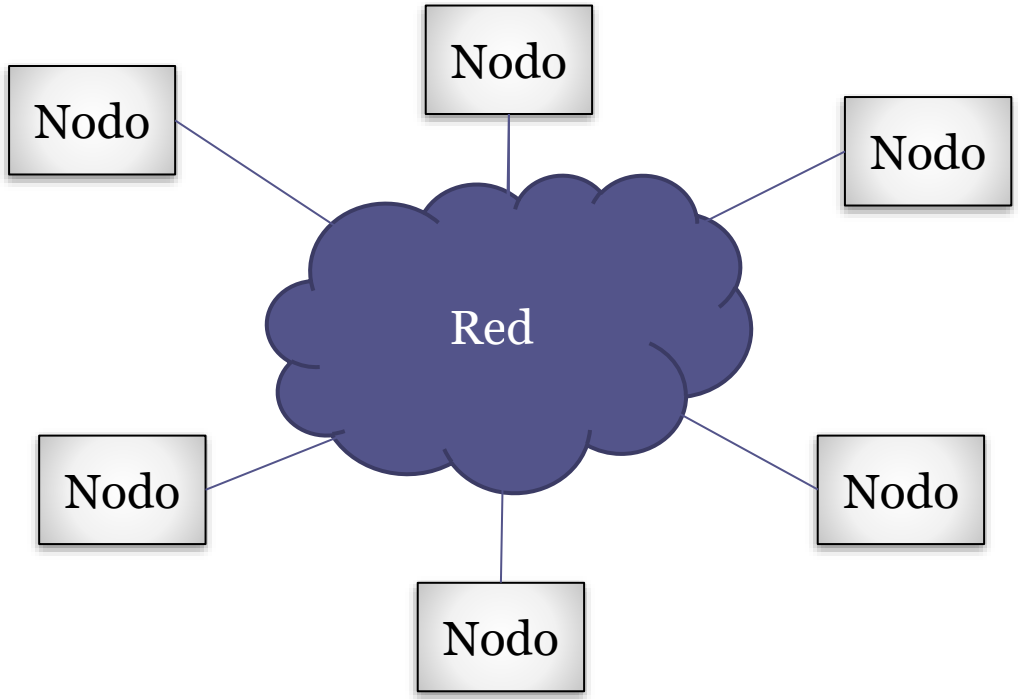
Patrón Bróker

Aplicaciones

- CORBA y sistemas distribuidos
- Android utiliza variación de patrón Bróker

Peer-to-Peer

Nodos (*peers*) iguales y autónomos se comunican entre sí



Peer-to-Peer

- Elementos
  - Nodos computacionales: *peers*
    - Tienen su propio estado e hilo de control
  - Protocolo de red
- Restricción
  - No existe un nodo principal

Peer-to-Peer

Ventajas

- Información y control descentralizados
- Tolerancia a fallos
  - No hay un punto único de fallo
  - Fallo de un nodo único no es determinante

Problemas

- Mantenimiento del estado del sistema
  - Complejidad de protocolo
- Limitaciones de ancho de banda
  - Latencia de la red y protocolo
- Seguridad
  - Detección de *peers* maliciosos

Peer-to-Peer

- Aplicaciones populares
  - Napster, BitTorrent, Gnutella, ...
  - No sólo compartir ficheros
    - Comercio electrónico (B2B)
    - Sistemas colaborativos
    - Redes de sensores
    - Blockchain
    - ...
- Variantes
  - Super-peers

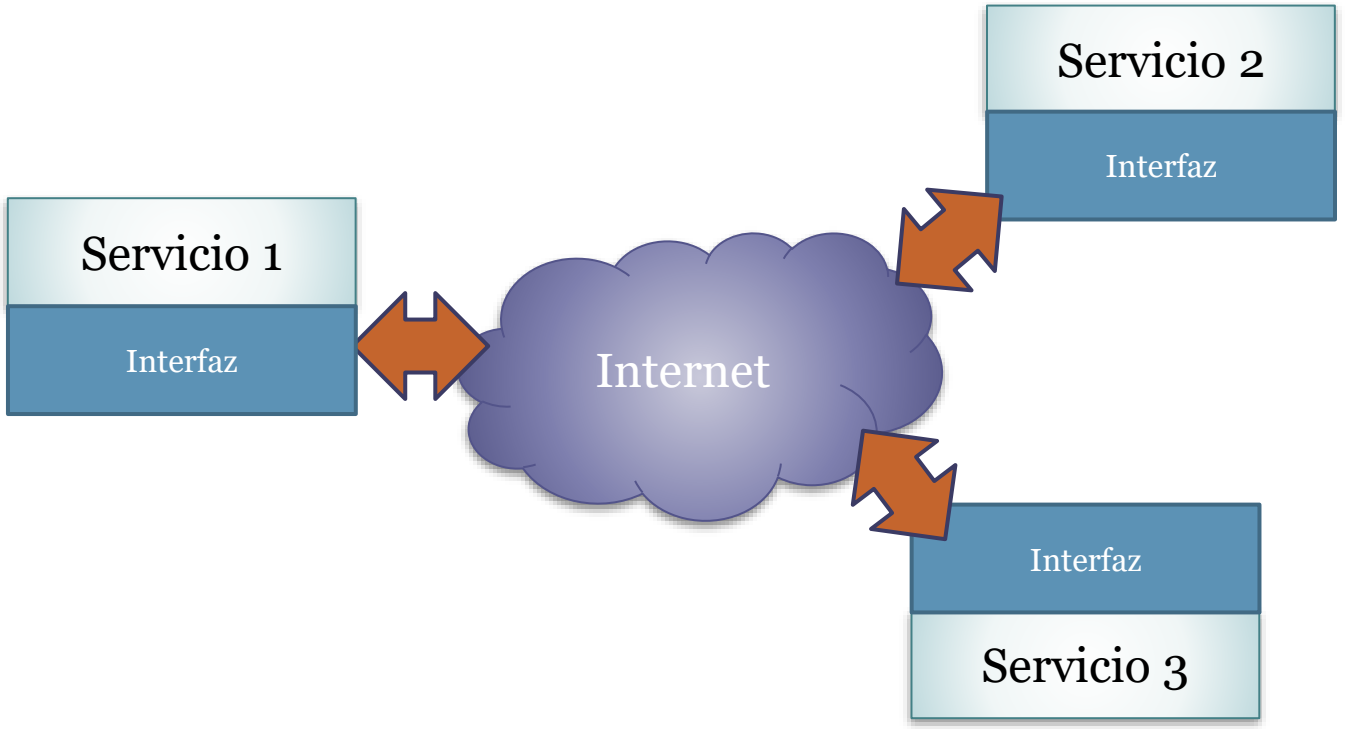
Servicios

- SOA - Service Oriented Architectures
  - WS-*
  - REST
  - Arquitecturas basadas en servicios
- Microservicios
- Serverless

SOA

SOA = Service Oriented Architecture

Los servicios están definidos mediante un interfaz



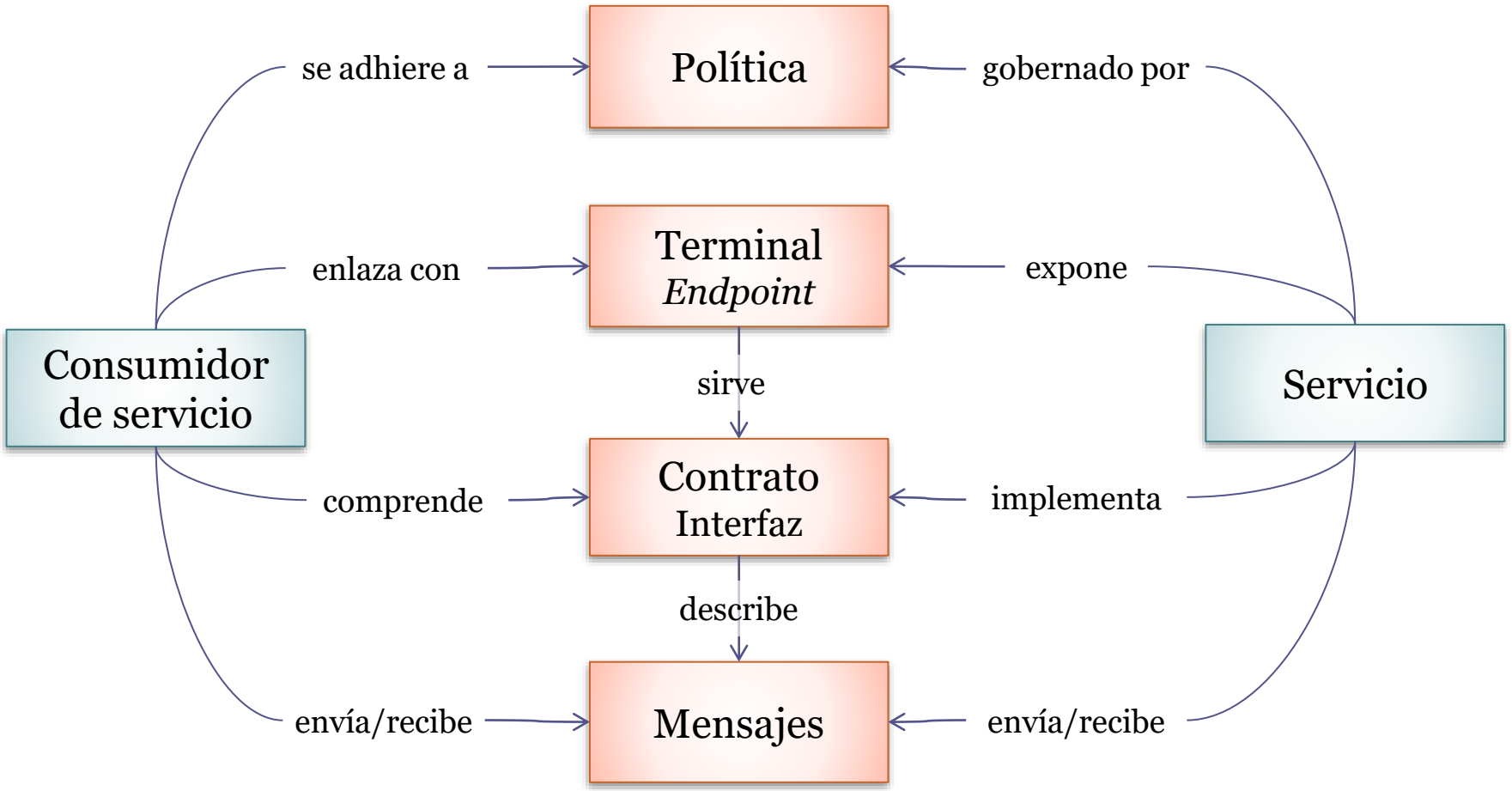
SOA

Elementos

- Proveedor : Proporciona el servicio
- Consumidor: Realiza peticiones al servicio
- Mensajes: Información intercambiada
- Contrato o interfaz: Descripción de la funcioanlidad ofrecida por el servicio
- Terminal: Ubicación del servicio
- Política: Acuerdos de gobierno del servicio
  - Seguridad, rendimiento, etc.

SOA

Restriccciones



Fuente: SOA Patterns, A. Rottem Gal Oz

SOA

Ventajas

- Independencia de lenguaje y plataforma
- Interoperabilidad
  - Utilización de estándares
- Acoplamiento débil
- Descentralizado
- Reusabilidad
- Escalabilidad
  - Comunicación uno-a-uno frente uno-a-muchos
- Mantenimiento sistemas *legacy*
  - Añadir una capa de servicios web

Problemas

- Eficiencia.
  - Puede no ser necesario en:
    - Entornos muy homogéneos, tiempo real, ...
- Exposición abierta
  - Riesgo de exponer API al exterior
- Seguridad
- Composición de servicios

SOA

Variantes:

- WS-*
- REST

WS-*

Modelo WS-* = Conjunto de especificaciones

- SOAP, WSDL, UDDI, etc....
- Propuesto por W3c, OASIS, WS-I, etc.
- Objetivo: Implementación SOA de referencia

Web Services Standards

Deutsche Post

Deutsche Post AG
SOP Group
Sträterchenweg 10
53113 Bonn

Phone +49 (2 28) 182 19 019
Fax +49 (2 28) 182 19 099
SOP.Group@DeutschePost.de
www.SOP-Group.com

Interoperability Issues

Business Process Specifications

Management Specifications

Metadata Specifications

Reliability Specifications

Security Specifications

Transaction Specifications

Resource Specifications

Messaging Specifications

SOAP

XML Specifications

Standards Bodies

Dependencies

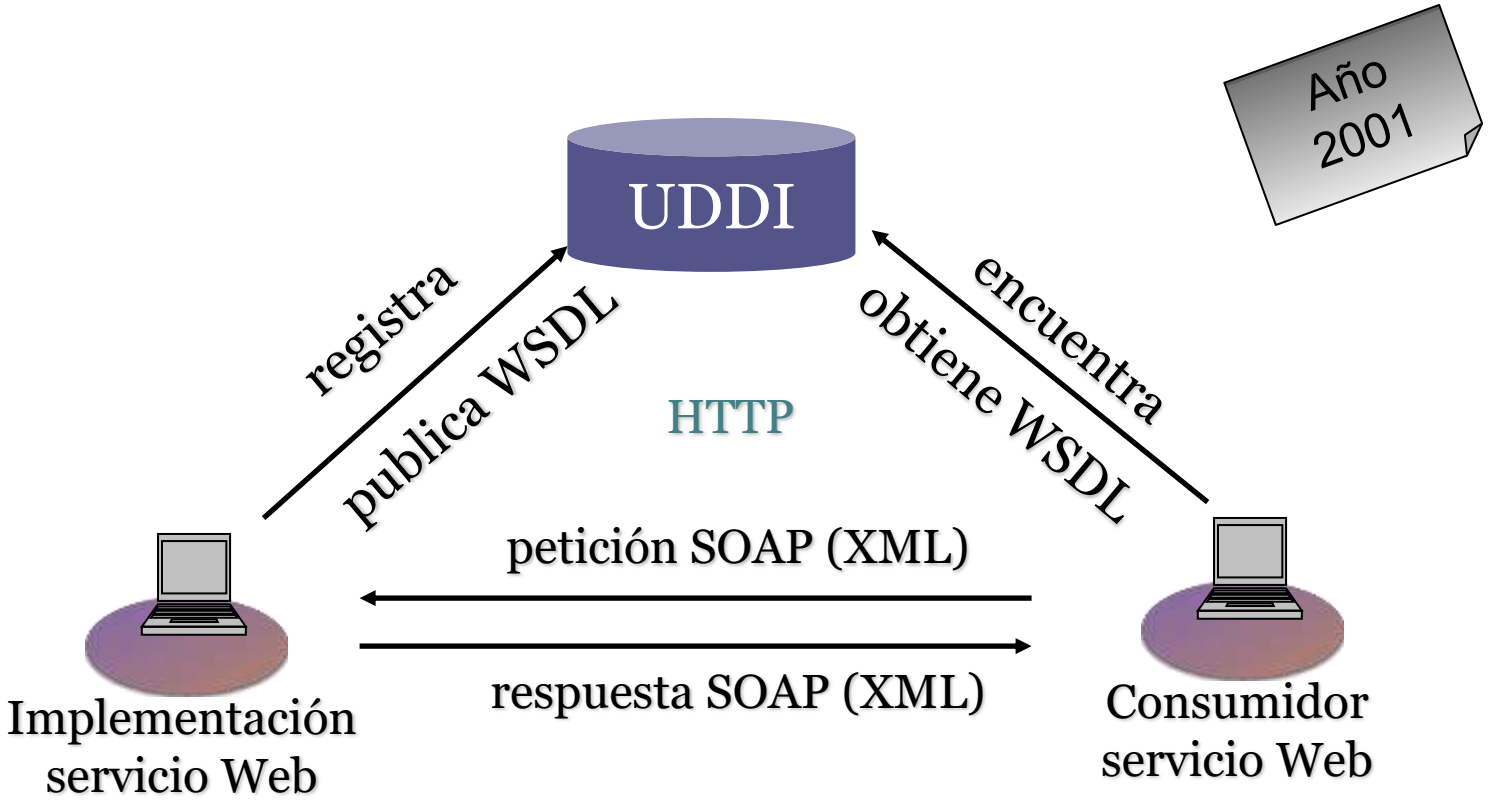
innoQ

innoQ Deutschland GmbH
Halskestraße 13
D-40880 Ratingen
Telefon +49 (0) 2102 - 77 162 - 100
Telefax +49 (0) 2102 - 77 162 - 199
info@innoq.com | www.innoq.com

innoQ Schweiz GmbH
Gewerbestrasse 11
CH-6330 Cham
Telefon +41 (0) 41 - 743 01 11
Telefax +41 (0) 41 - 743 01 19

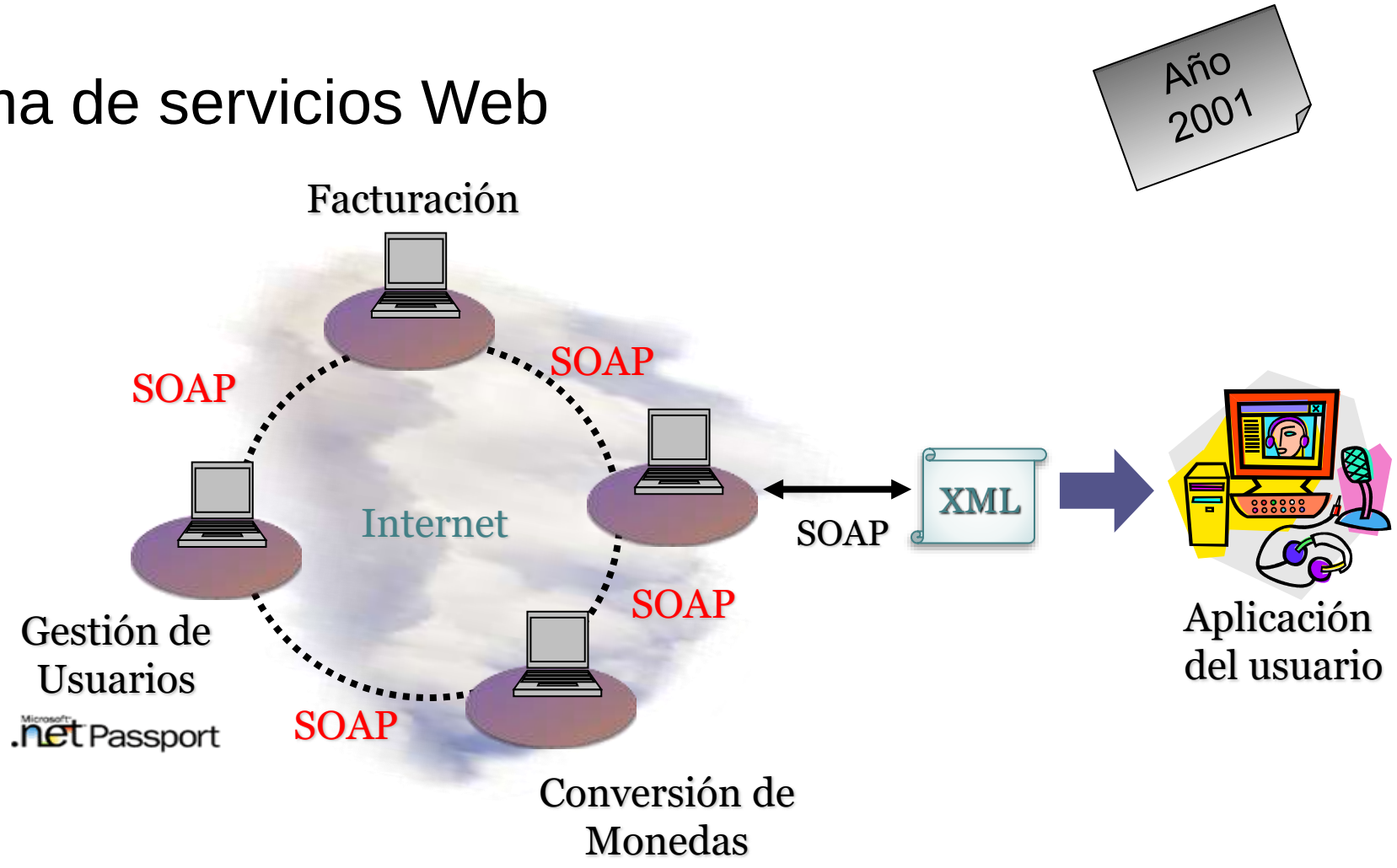
Version 2.0 - September 2005

WS-*


Año 2001
UDDI
registra
publica WSDL
encuentra
obtiene WSDL
HTTP
petición SOAP (XML)
respuesta SOAP (XML)
Implementación servicio Web
Consumidor servicio Web

WS-*

Ecosistema de servicios Web



WS-*

SOAP

Define el formato de los mensajes y varios enlaces con protocolos

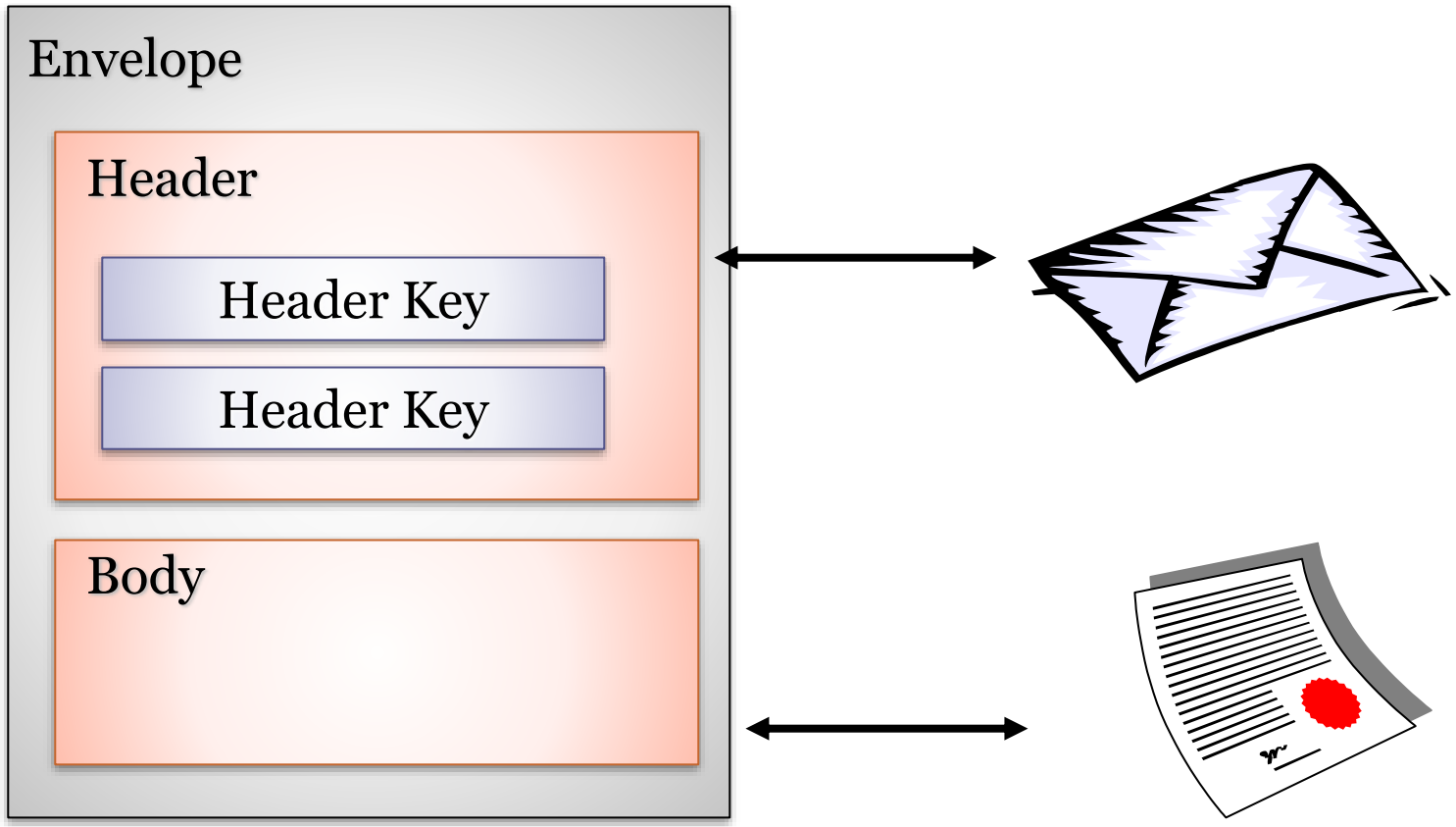
- Originalmente *Simple Object Access Protocol*

Evolución

- Desarrollado a partir de XML-RPC
- SOAP 1.0 (1999), 1.1 (2000), 1.2 (2007)
- Participación inicial de Microsoft
- Adopción posterior de IBM, Sun, etc.
- *Bastante* aceptación industrial

WS-*

Esquema de SOAP



WS-*

Ejemplo de SOAP sobre HTTP

Año 2001

POST ?

```
POST /Suma/Service1.asmx HTTP/1.1
Host: localhost
Content-Type: text/xml; charset=utf-8
Content-Length: longitod del mensaje
SOAPAction: "http://tempuri.org/suma"
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope
     xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
<soap:Body>
  <suma xmlns="http://tempuri.org/">
     <a>3</a>
     <b>2</b>
  </suma>
 </soap:Body>
</soap:Envelope>
```

WS-*

Ventajas

- Especificaciones realizadas por comunidad
  - W3c, OASIS, etc.
- Adopción industrial
  - Implementaciones
- Visión integral de servicios web
- Numerosas extensiones:
  - Seguridad, orquestación, coreografía, etc.

Problemas

- No todas las especificaciones están maduras
  - Sobre-especificación
  - Falta de implementaciones
- Abuso del estilo RPC
  - Interfaz no uniforme
  - No se sigue arquitectura HTTP
    - Métodos GET/POST sobrecargados

WS-*

SOAP en la práctica

- Numerosas aplicaciones utilizan SOAP
  - Ejemplo: eBay (50mill. transacciones SOAP al día)
- Pero…algunos servicios web populares dejaron de ofrecer soporte SOAP
  - Ejemplos: Amazon, Google, etc.

REST

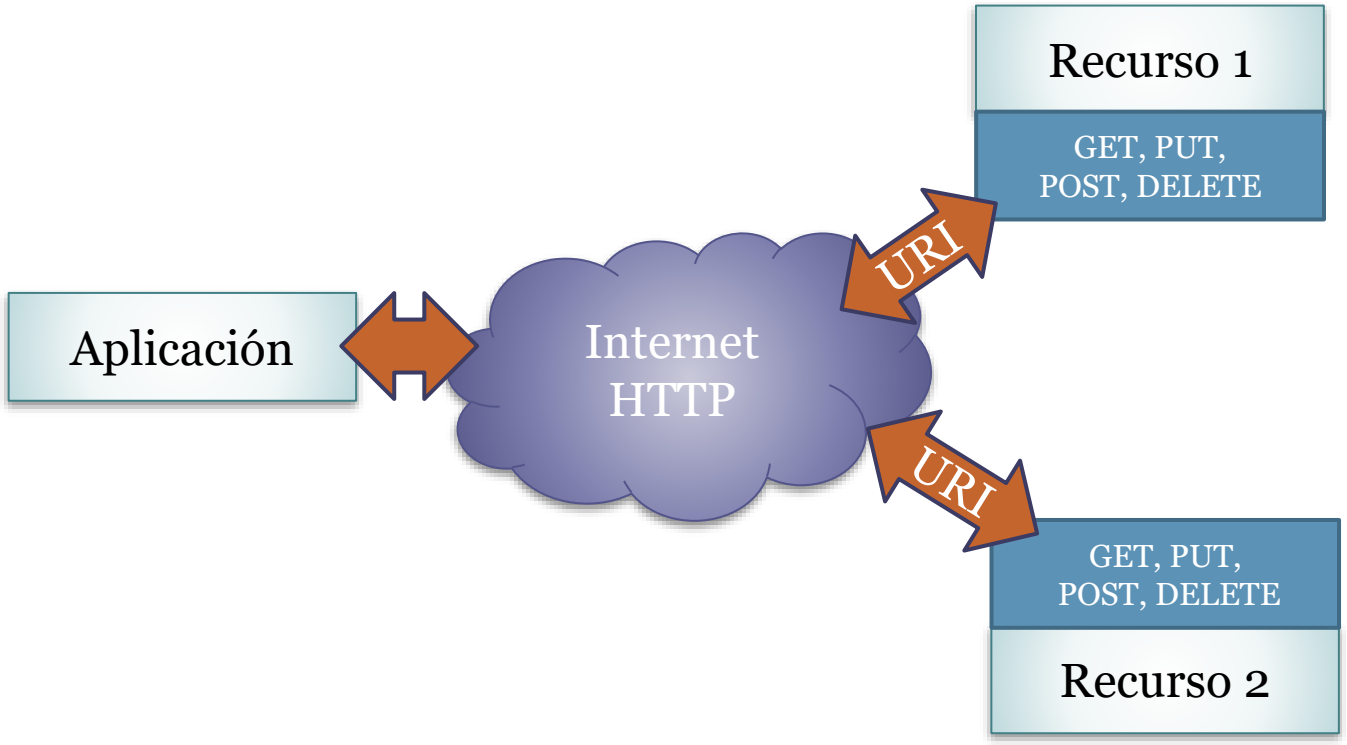
REST = REpresentational State Transfer

- Estilo de arquitectura
- Origen: Tesis doctoral de Roy T Fielding (2000)
- Inspirado en la arquitectura de la Web (HTTP/1.1)

REST

REST - Representational State Transfer

Transferencia de representación de estado



REST

- Conjunto de restricciones
  - Recursos con interfaz uniforme
    - Identificables mediante URIs
  - Se devuelven representaciones de los recursos
  - Sin estado
  - Interfaces genéricos
    - Conjunto acciones: GET, PUT, POST, DELETE
- REST = estilo de arquitectura
  - Varios niveles de adopción:
    - RESTful
    - Híbrido REST-RPC

REST

- En capas
- Cliente-servidor
  - Sin estado
  - Caché
  - Servidor replicado
- Interfaz uniforme
  - Identificación de recursos (URIs)
  - Manipulación de representaciones de recursos
  - Mensajes auto-descriptivos (tipos MIME)
  - Enlaces a otros recursos (HATEOAS)
- Código bajo demanda (opcional)

REST

Interfaz uniforme:

- Conjunto de operaciones limitado
  - GET, PUT, POST, DELETE
- Conjunto limitado de tipos de contenidos
  - En HTTP se identifican mediante tipos MIME: XML , HTML...

Método	En Bases de datos	Función	Segura?	Idempotente?
PUT	≈Create/Update	Crear/actualizar	No	SI
POST	≈Update	Crea/actualiza subordinado	No	No
GET	Retrieve	Consultar recurso	Si	Si
DELETE	Delete	Eliminar recurso	No	Si

Seguro = No modifica los datos del servidor
Idempotente = El efecto de ejecutarlo n-veces es el mismo que el de ejecutarlo 1 vez

REST

- Protocolo cliente/servidor sin estado
- Estado gestionado por el cliente
  - HATEOAS *(Hypermedia As The Engine of Application State)*
    - *Respuestas devuelven URIs a opciones disponibles*
    - Peticiones sucesivas de recursos

Ejemplo: Gestión de alumnos
1.- Obtener lista de alumnos
`GET http://ejemplo.com/alumnos`
Devuelve lista de URIs de alumnos
2.- Obtener información de ese alumno
`GET http://ejemplo.com/alumnos/id2324`
3.- Actualizar información de ese alumno
`PUT http://ejemplo.com/alumnos/id2324`

REST

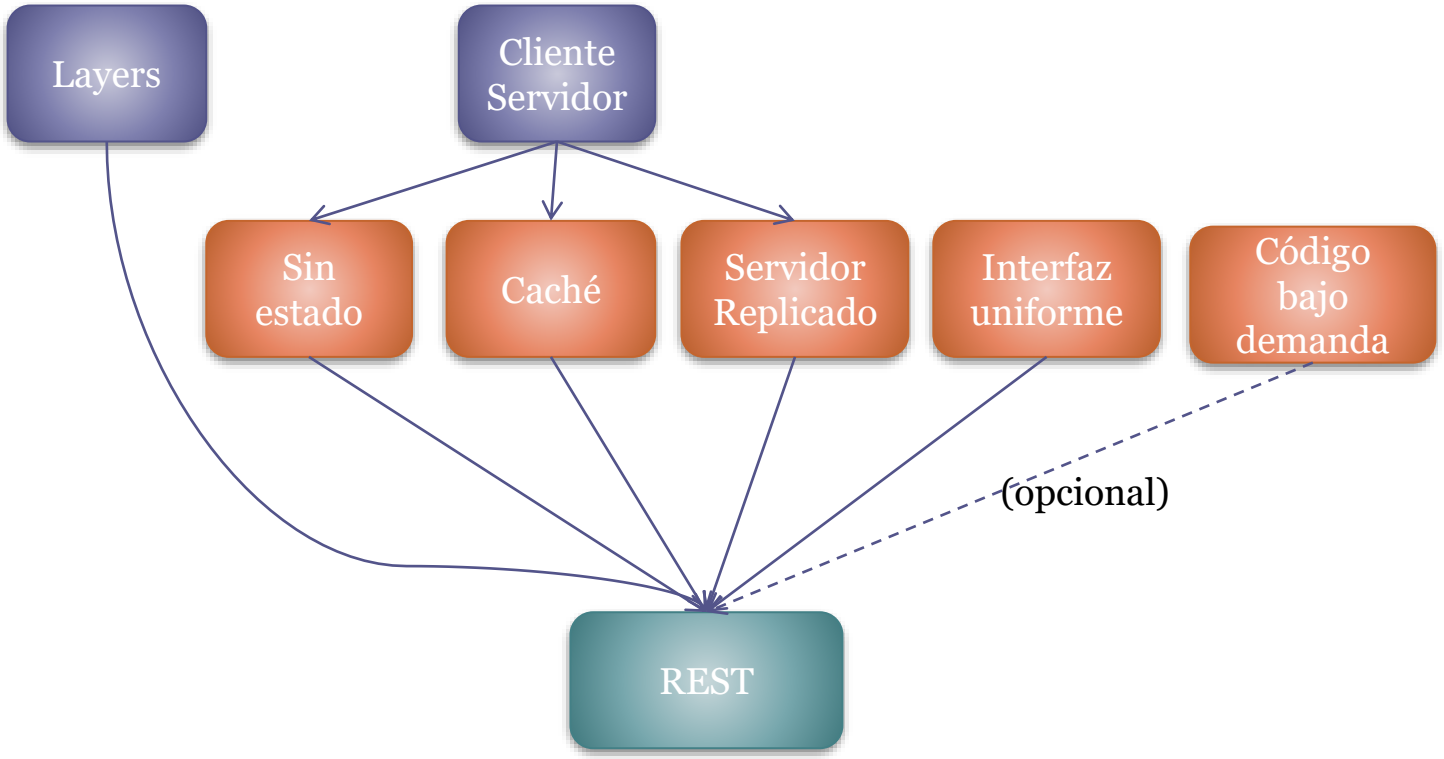
Ventajas

- Cliente/Servidor
  - Separación de responsabilidades
  - Acoplamiento débil
- Interfaz uniforme
  - Facilita comprensión
  - Desarrollo independiente
- Escalabilidad
- Mejora tiempos de respuesta
- Menor carga en red (caché)
  - Ahorro de ancho de banda

Problemas

- REST mal entendido
  - Uso de JSON o XML sin más
    - Servicios Web sin contrato ni descripción
  - REST con estilo RPC
- Dificultades para incorporar otros requisitos
  - Seguridad, transacciones, composición, etc.

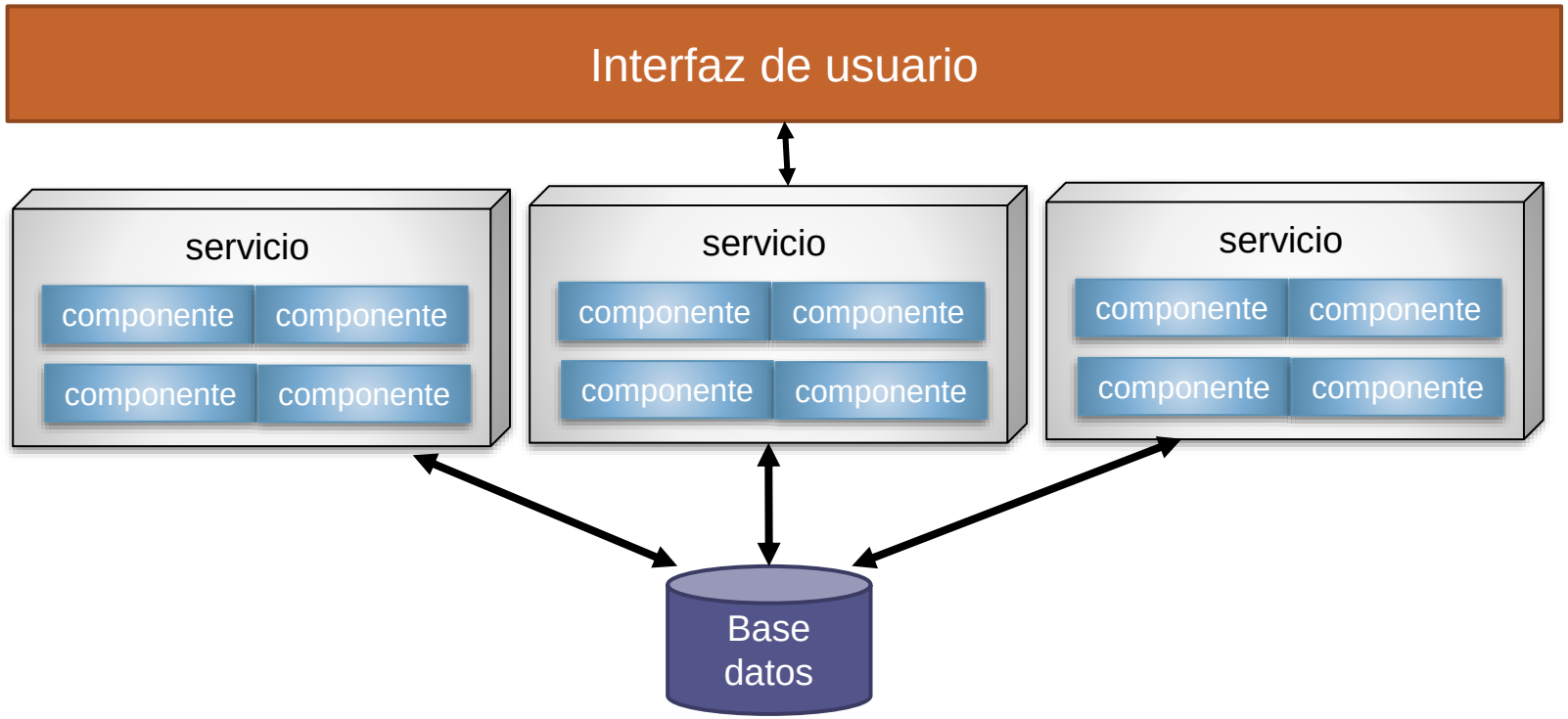
REST como estilo compuesto



Arquitectura basada en servicios

Estilo arquitectónico pragmático basado en SOA

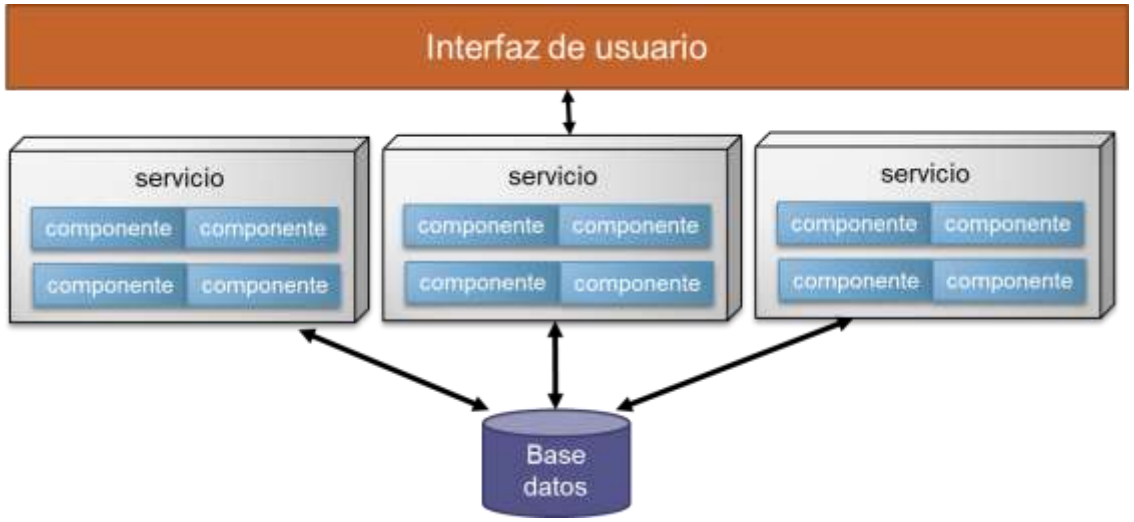
Una de las implementaciones más habituales



Arquitectura basada en servicios

Elementos

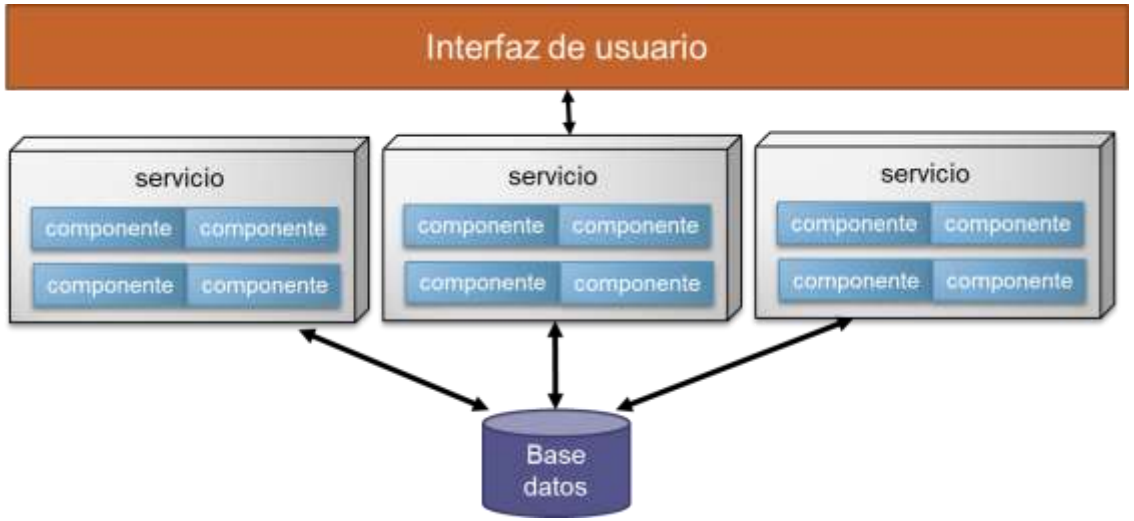
- Servicios = Unidades desplegadas independientemente
  - Normalmente formados por varios componentes
- El interfaz de usuario accede a servicios de forma remota (Internet)
- Base de datos



Arquitectura basada en servicios

Restricciones

- Cada servicio es desplegado de forma independiente
- Servicios pueden ser grandes
- Interfaz de usuario puede dividirse (varias topologías)
- Base de datos compartida por cada servicio



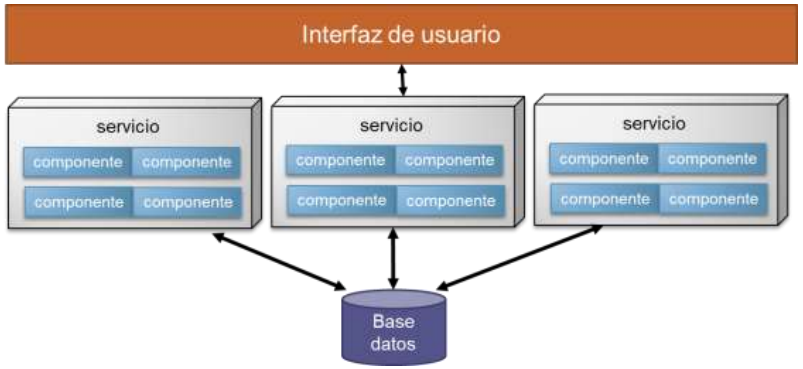
Arquitectura basada en servicios

Ventajas

- Modularidad de desarrollo
  - Servicios independientes
- Diversidad tecnológica
  - Cada servicio puede implementarse con tecnologías y lenguajes diferentes
- *Time to market*
  - Varios *frameworks* disponibles
- Disponibilidad
- Fiabilidad

Retos

- Escalabilidad (particionado base datos)
- Evolución de servicios
  - Difícil adaptación al cambio
  - Servicios como monolitos
- Ley de Conway
  - Equipos de base de datos, interfaz de usuario, desarrolladores...

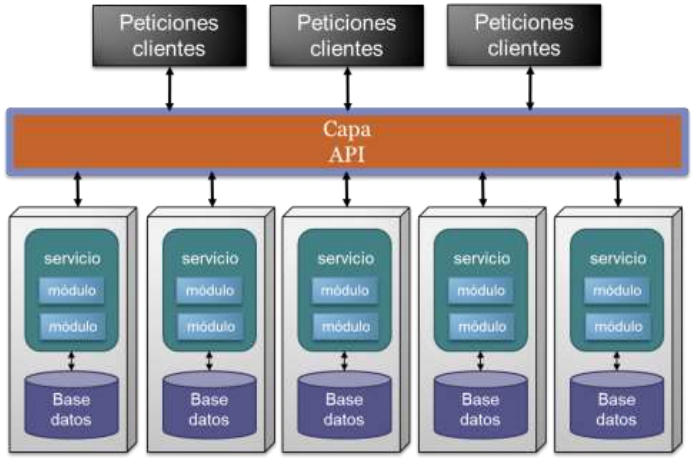


Microservicios

Descomponer aplicaciones en microservicios

Cada microservicio = bloque independiente de construcción/despliegue

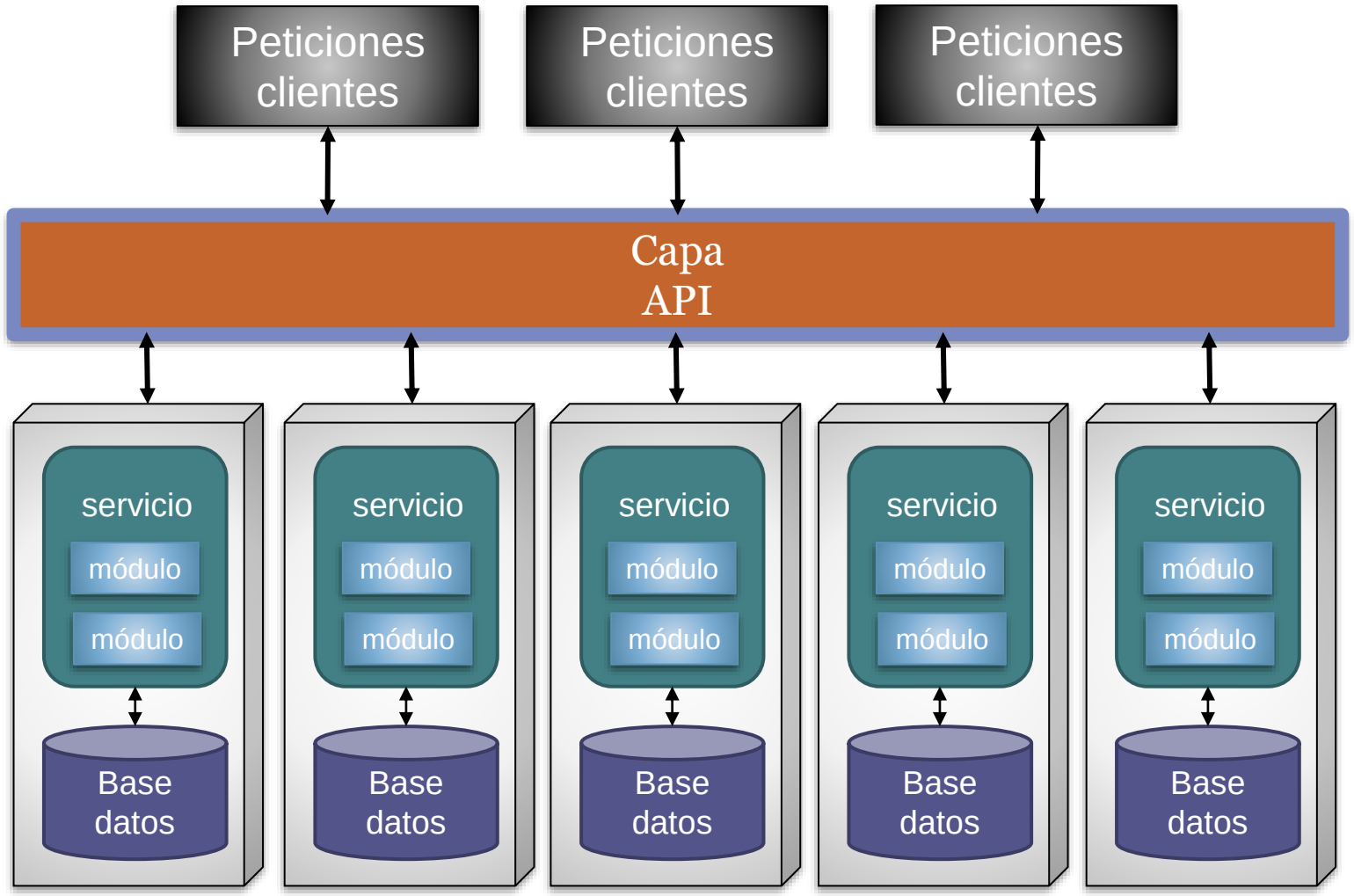
- Altamente desacoplados
- Enfocados a hacer una tarea bien definida
- Gestionan sus datos



http://martinfowler.com/articles/microservices.html

Microservicios

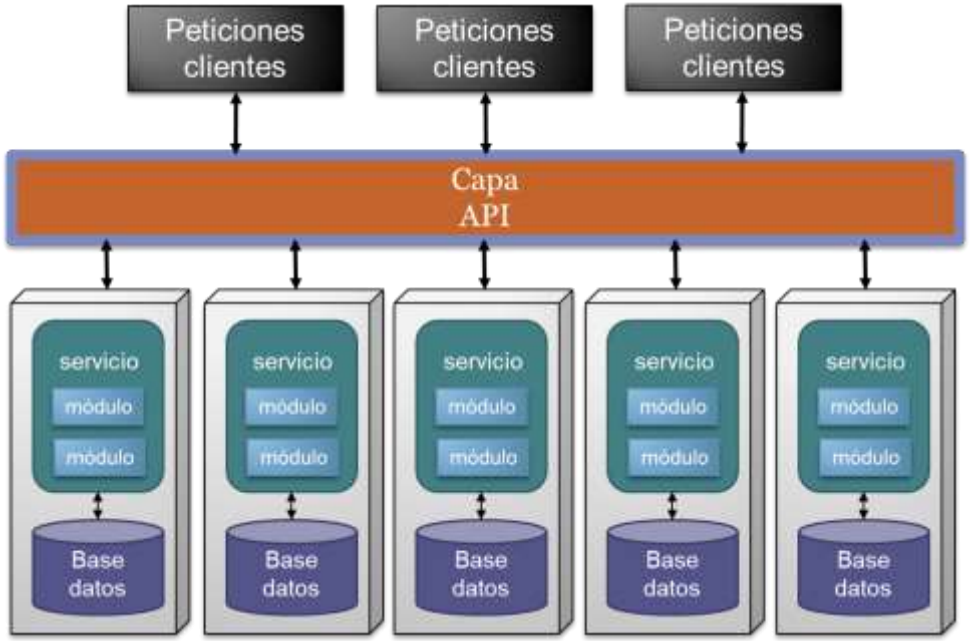
Diagrama



Microservicios

Elementos

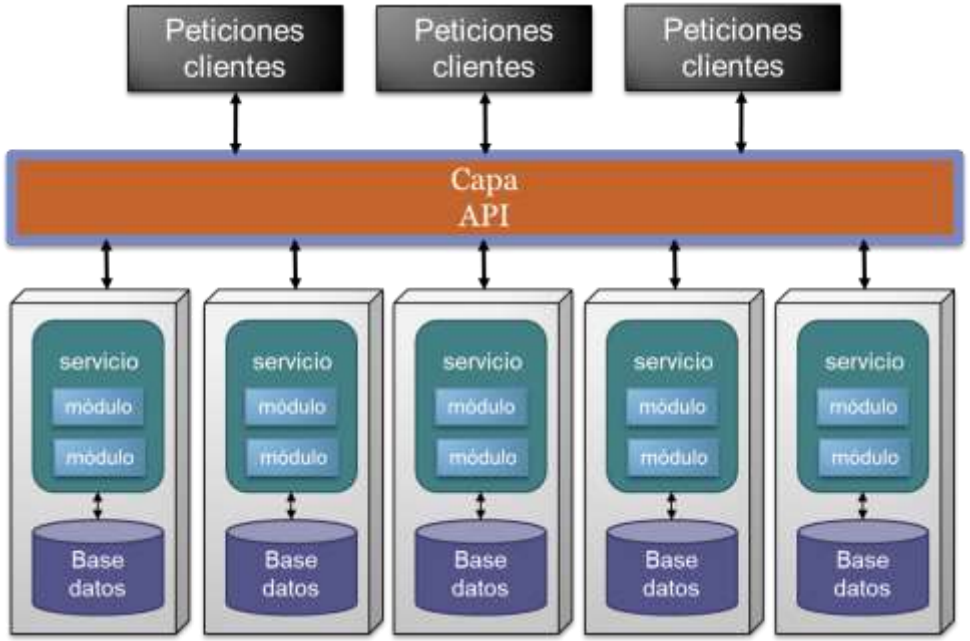
- Componente desplegado = Servicio + base datos
- Servicio puede contener varios módulos
- Capa API (opcional) es un proxy o servicio de nombrado



Microservicios

Restricciones

- Servicios distribuidos
- Contexto definido (*bounded context*):
  - Cada servicio modela un dominio o flujo de trabajo
- Aislamiento de datos
- Independencia:
  - No hay mediador u orquestador



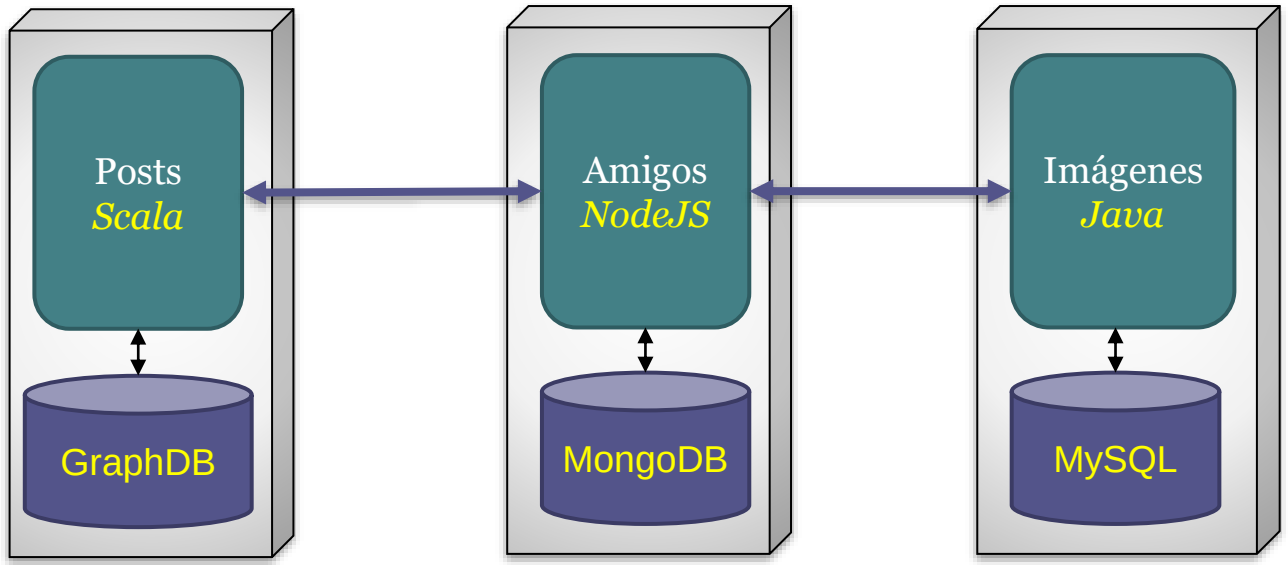
Características/ventajas

- Diversidad tecnológica
- Resiliencia
- Escalabilidad bajo demanda
- Desplegabilidad
- Alineación de la organización
- Gestión descentralizada de datos
- Optimización para sustitución

Microservicios y diversidad

Cada microservicio puede implementarse en su propio lenguaje de programación y pila tecnológica

- Facilita experimentación con nuevas tecnologías
- Flexibilidad



Resiliencia

Si un componente de un sistema falla y el fallo no escala, el sistema puede seguir funcionando

En un sistema monolito, si un componente falla, todo el sistema deja de funcionar

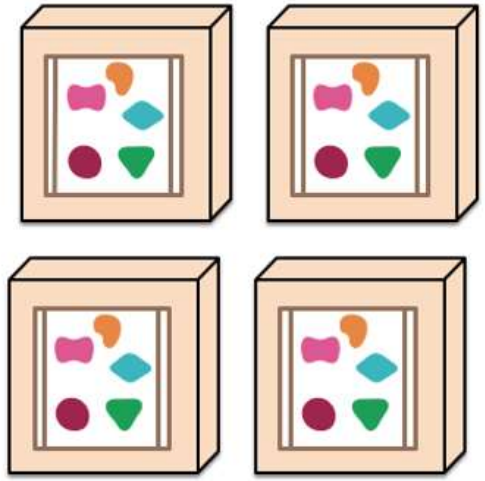
Escalabilidad

- Es posible escalar bajo demanda ciertos microservicios
  - Los sistemas monolito requieren escalar todo el sistema de forma homogénea
  - No todos los componentes tienen las mismas necesidades de escalabilidad
    - Los microservicios pueden ser replicadas según sea necesario

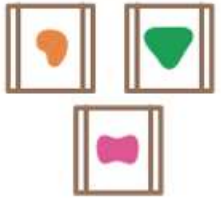
Escalabilidad

Aplicación monolítica
Toda funcionalidad en un solo proceso

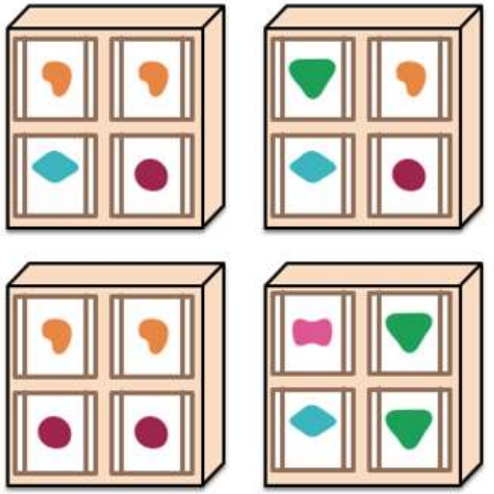
...escalabilidad mediante replicación del monolito en diferentes servidores



Microservicios: Cada funcionalidad distribuida en un microservicio



...escalabilidad mediante distribución de servicios en servidores y replicación



Desplegabilidad

- Puede despegarse cada servicio de forma independiente
  - Permite realizar un cambio en un servicio y desplegarlo inmediatamente
  - Facilita el despliegue continuo

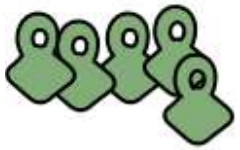
Alineación de la organización

- Maniobra inversa de Ley de Conway
  - Evolucionar los equipos y las estructuras organizativas para promover la arquitectura deseada
- Crear equipos siguiendo la descomposición modular
  - Equipos con funcionalidad cruzada
- Propiedad del servicio: el equipo que es dueño del servicio es responsable de realizar cambios y desplegarlo
  - "You build it, you run it" (Amazon)
- Objetivo: mayor autonomía y velocidad de despliegue

Aplicaciones tradicionales

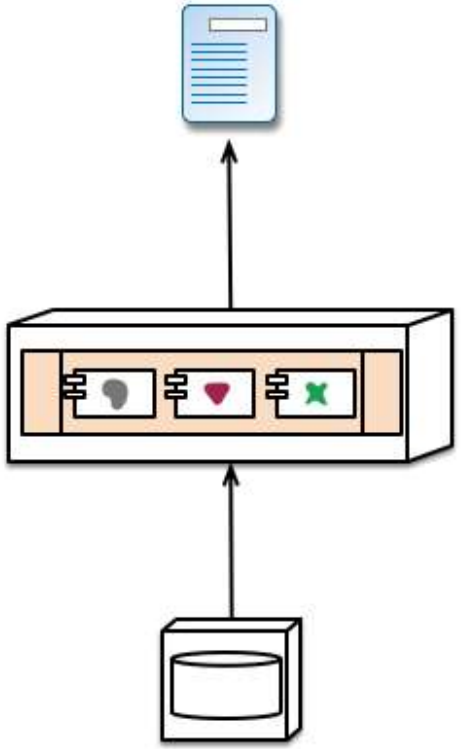
Especialistas Interfaz Usuario

Especialistas Middleware



DBAs

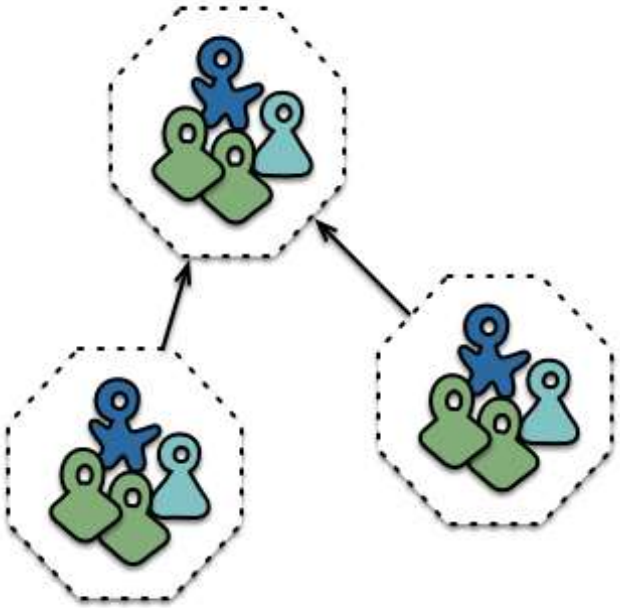
Equipos especializados (en silos)



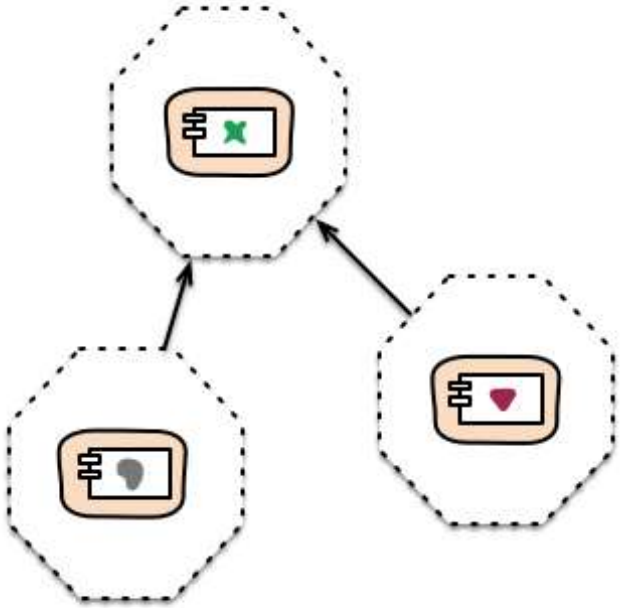
...lleva a arquitecturas basadas en silos
Debido a la Ley de Conway

Con microservicios

Equipos basados en capacidades



Equipos multidisciplinares
Funcionalidad cruzada

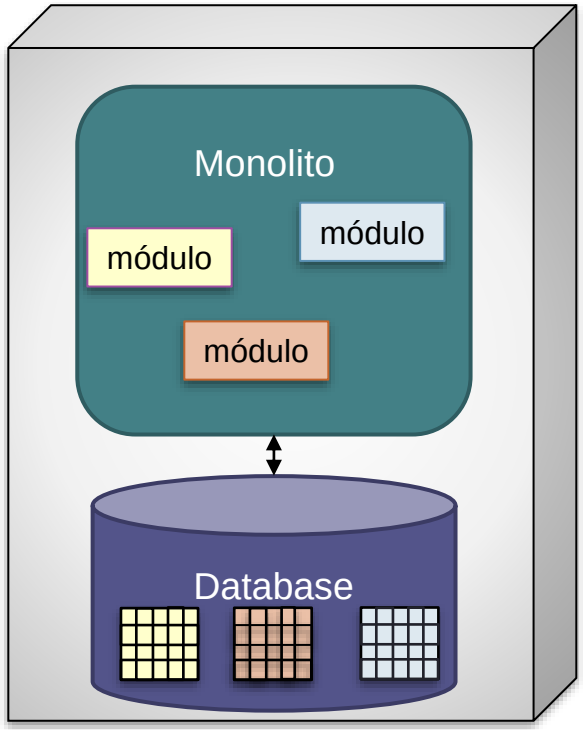


Organizados alrededor de las capacidades
Debido a la Ley de Conway

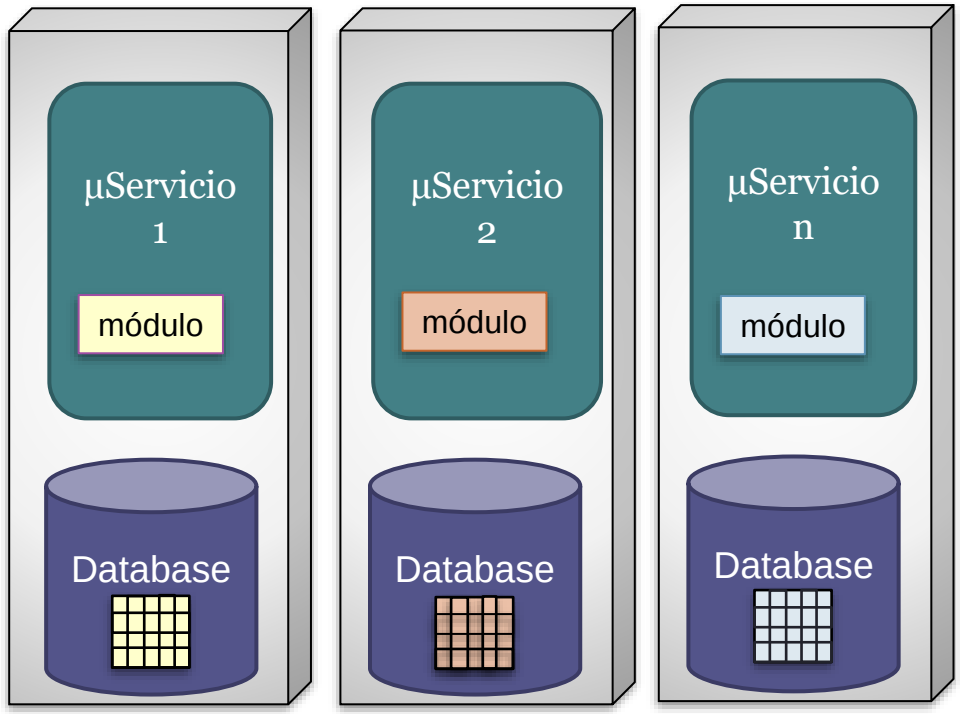
Gestión de datos descentralizada

Cada equipo/servicio gestiona sus datos

Monolito - única base de datos



Microservicios - Bases de datos propias



Optimización para sustitución

- Sistemas tradicionales normalmente contienen sistemas antiguos (legacy) que nadie se atreve a tocar
- Con microservicios
  - Menor coste para sustituir un microservicio por una mejor implementación
  - O incluso borrarlo

Retos de los microservicios

- Gestión de muchos microservicios
  - Demasiados microservicios = antipatrón (nanoservicios)
  - Garantizar la consistencia de la aplicación
- Complejidad de desarrollo
  - Sistemas distribuidos son difíciles de gestionar
  - Aparecen nuevos problemas: latencia, formato de mensajes, balance de carga, tolerancia a fallos, etc.
- Pruebas y despliegue
  - Complejidad operacional
- Antipatrón: monolito distribuido
  - Microservicios enmarañados que no son independientes
- Deterioro estructural (*ver siguiente trasparencia*)

http://martinfowler.com/articles/microservice-trade-offs.html
https://www.ufried.com/blog/microservices_fallacy_1/

Deterioro estructural microservicios

Dependencias de código entre microservicios

- Demasiadas librerías compartidas

Demasiada comunicación entre servicios

Demasiadas peticiones de orquestación

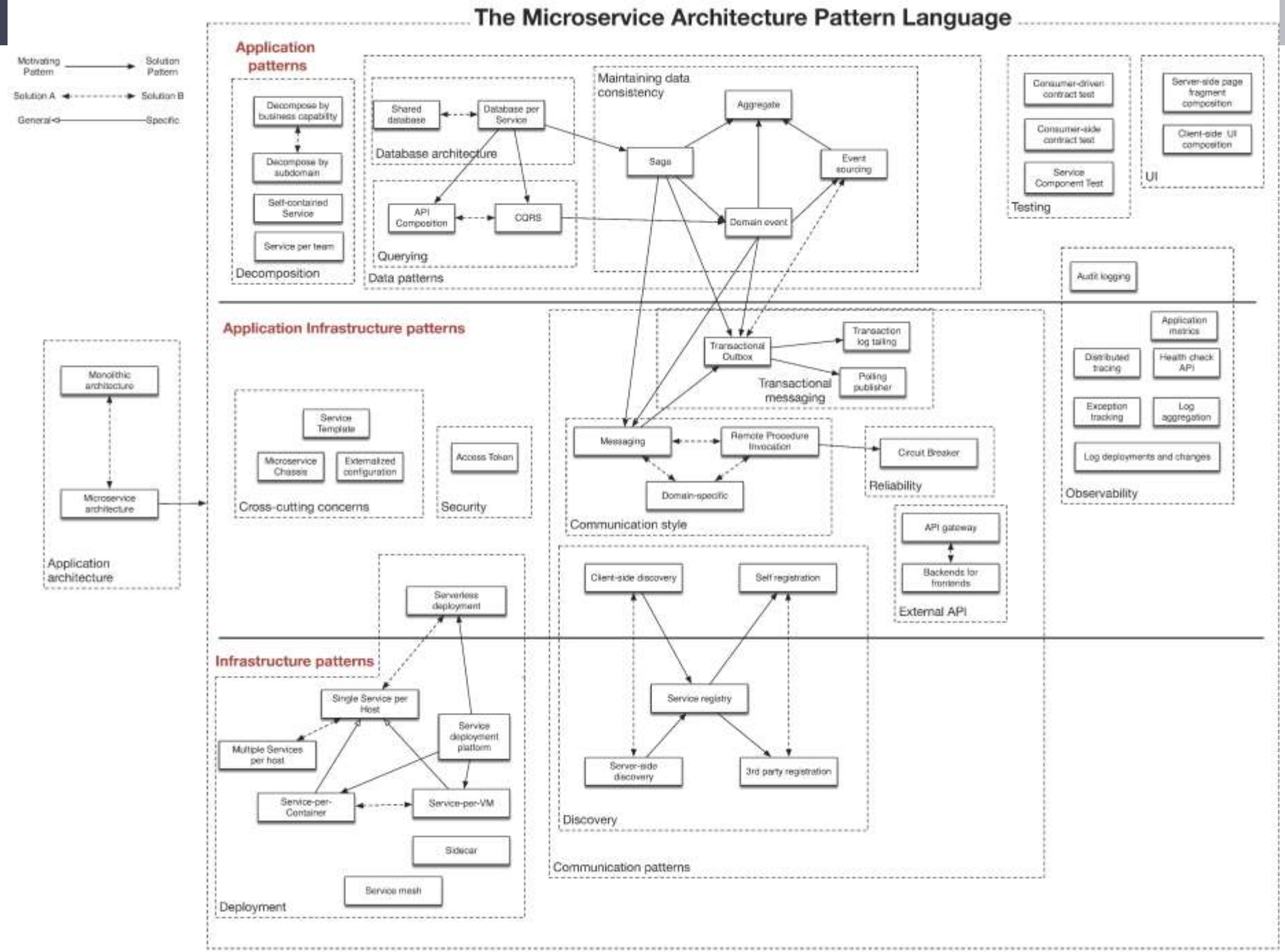
Agregación de microservicios

Acoplamiento de la base de datos

Antipatrón: Monolito distribuido

Vídeo: Analyzing architecture (microservices)
https://www.youtube.com/watch?v=U7s7Hb6GZCU

Patrones Microservicios



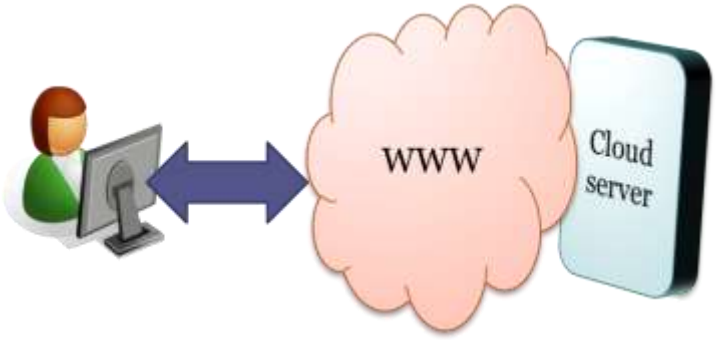
Microservicios

Variantes

Arquitectura de sistemas auto-contenidos

- Self contained Systems (SCS) Architecture
- Separación de funcionalidad en muchos sistemas independientes
- https://scs-architecture.org/
- Cada Sistema auto-contenido contiene lógica y datos

Arquitectura *Serverless*



También conocido como:
- Function as a service (FaaS)
- Backend as a service (BaaS)

Aplicaciones dependen de servicios de terceras partes

Los desarrolladores no tienen que preocuparse de los servidores
- Escalabilidad automática

Clientes ricos
- *Aplicaciones Single Page, Aplicaciones móviles*

Ejemplos:
- AWS Lambda, Google Cloud Functions, Ms Azure Functions

https://en.wikipedia.org/wiki/Serverless_computing
https://martinfowler.com/articles/serverless.html

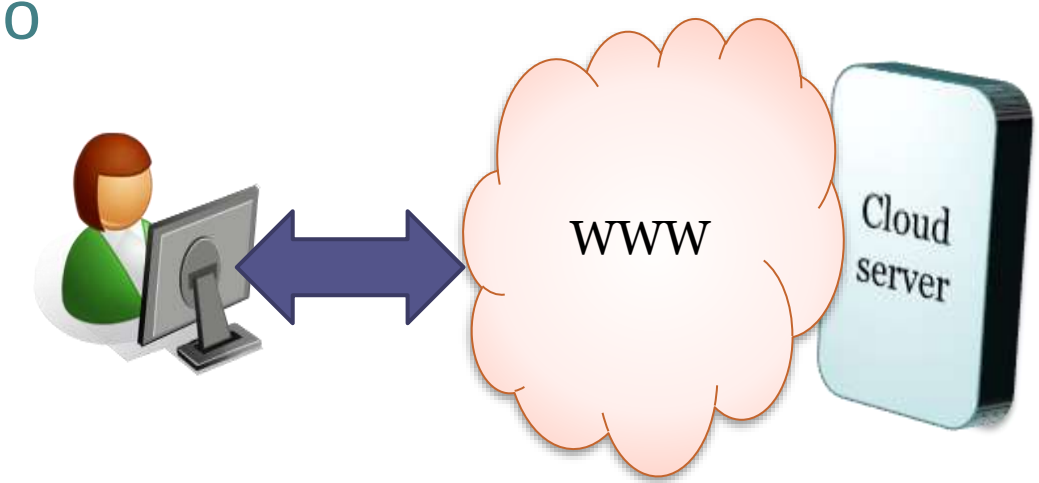
Serverless

Elementos

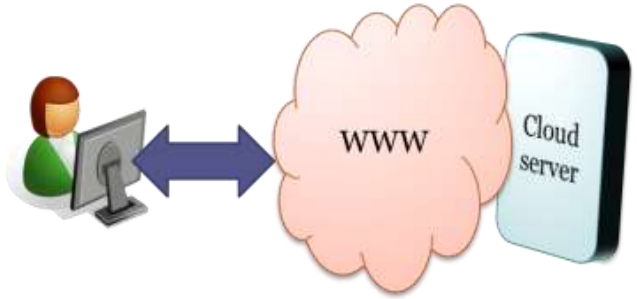
- Cliente ejecuta funciones como servicios
- Servidor en la nube proporciona *backend as a service*

Restricciones

- No se gestionan los servidores directamente
- Escalabilidad automática y suministro basado en carga
- Costes basados en uso preciso



Arquitectura *Serverless*



Ventajas

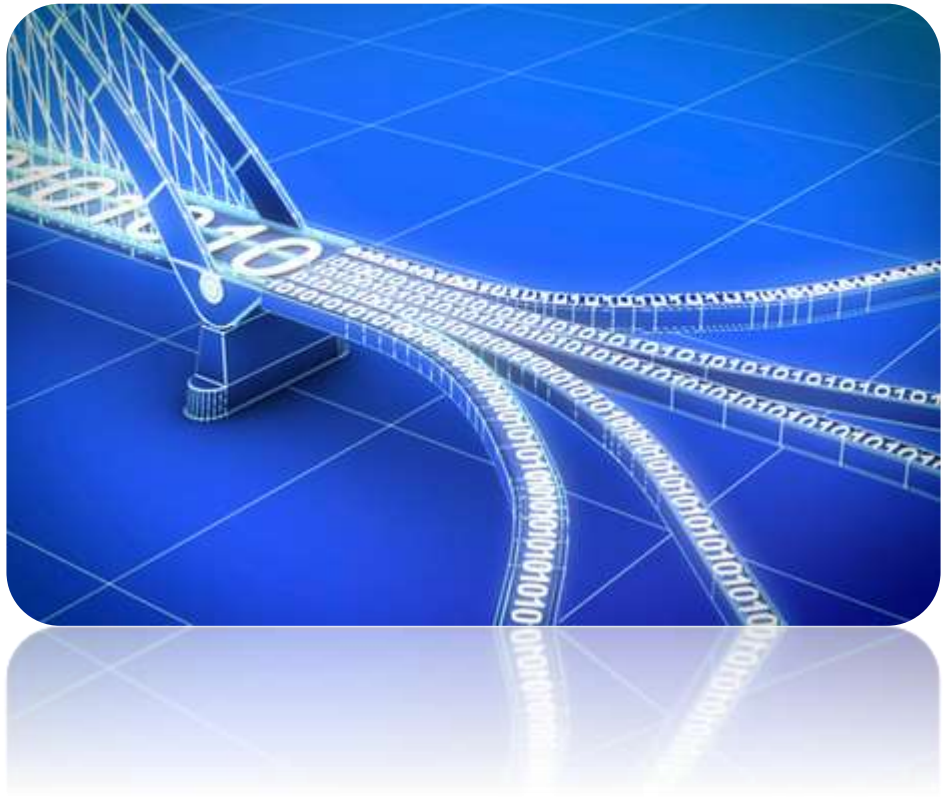
- Escalabilidad automática
- Alta disponibilidad implícita
- Rendimiento no se mide en términos de tamaño o coste de servidor
- Costes basados en uso
  - Sólo se paga por los recursos requeridos
- *Time to market* reducido

Retos

- Dependencia de un vendedor
  - Vendor lock-in
  - Incompatibilidad entre soluciones de diferentes vendedores
- Seguridad
- Latencia de arranque
- Pruebas de integración
- Monitorización/depuración

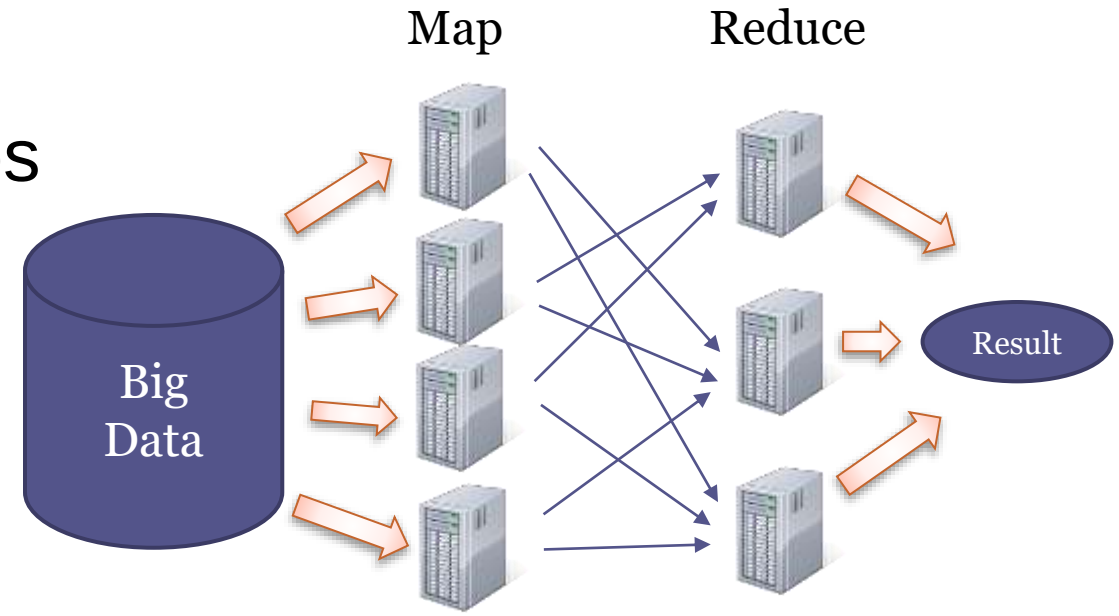
Sistemas escalables y big data

MapReduce
Arquitectura Lambda
Arquitectura Kappa



MapReduce

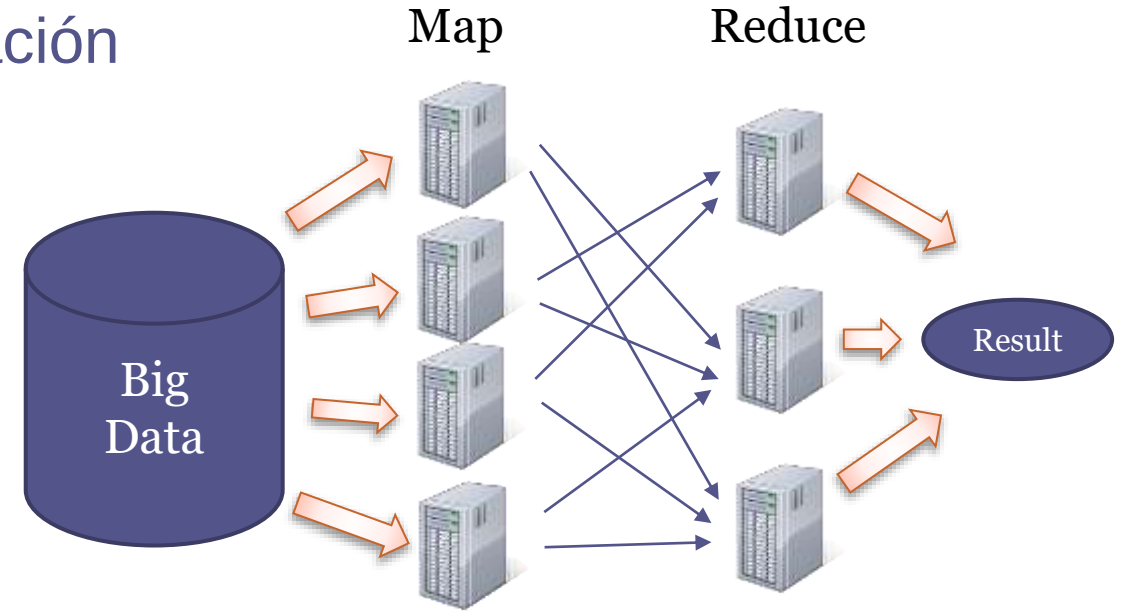
- Propuesto por Google
  - Publicado en 2004
  - Implementación interna propietaria
- Objetivo: grandes cantidades de datos
  - Muchos nodos computacionales
  - Tolerancia a fallos
- Estilo compuesto de
  - Master-slave
  - Secuencial (*batch*)



MapReduce

Elementos

- Nodo maestro: controla la ejecución
  - Gestiona sistema de ficheros con replicación
- Nodos esclavos
  - Ejecutan tareas *mapper* y *reducer*
  - Tolerancia a fallos de nodos
  - Hardware/software heterogéneo



MapReduce - Esquema

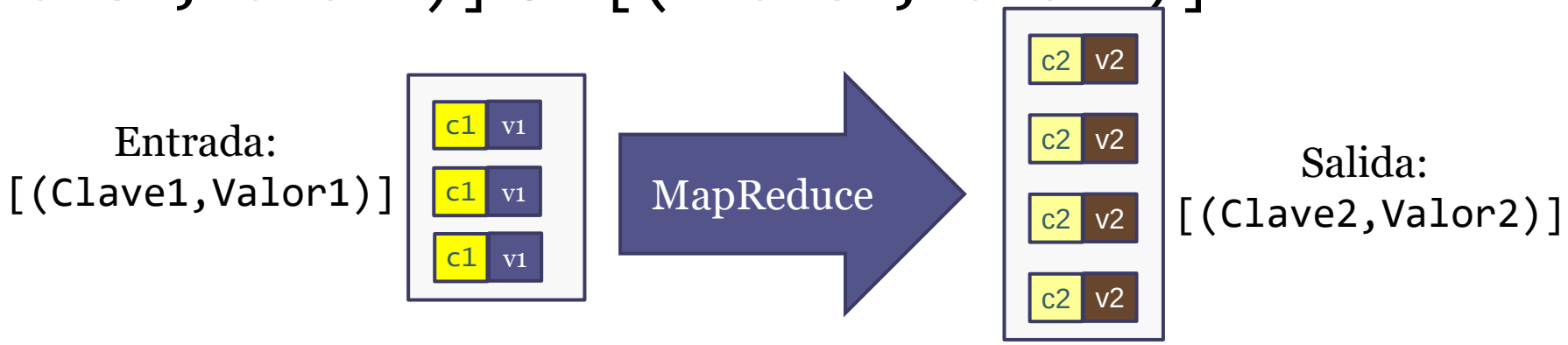
Inspirado en P. funcional:

- 2 componentes: mapper y reducer

Los datos se trocean para su procesamiento

Cada dato asociado a una clave

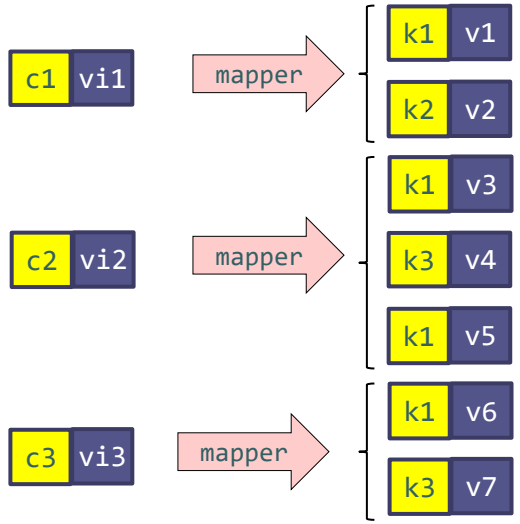
Transforma `[(clave1,valor1)]` en `[(clave2,valor2)]`



Paso 1 - Mapper

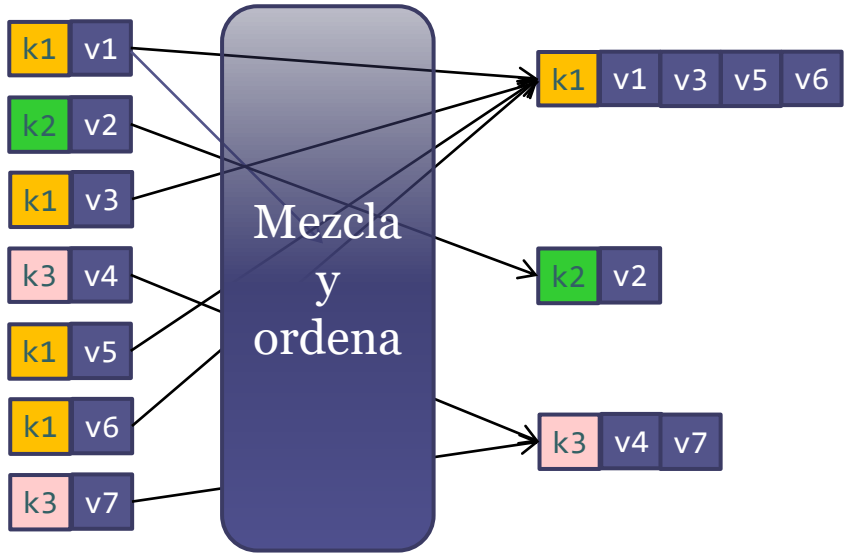
Para cada (clave1,valor1) devuelve una lista de (clave2,valor2)

Tipo: `(clave1, valor1)` → `[(clave2,valor2)]`



Paso 2 - Mezcla y ordenación

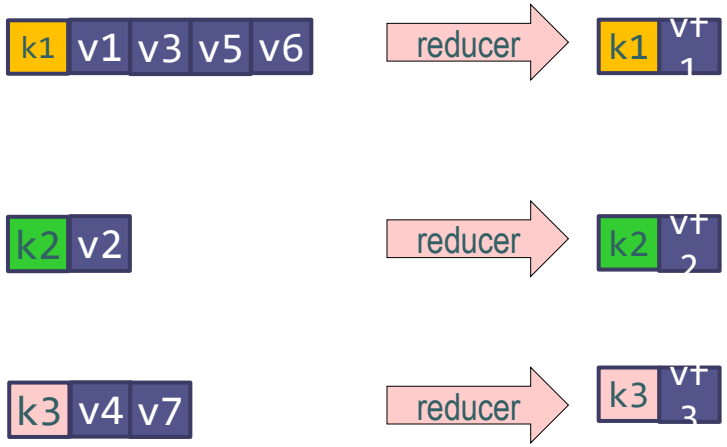
El sistema se encarga de mezclar y ordenar resultados intermedios en función de las claves



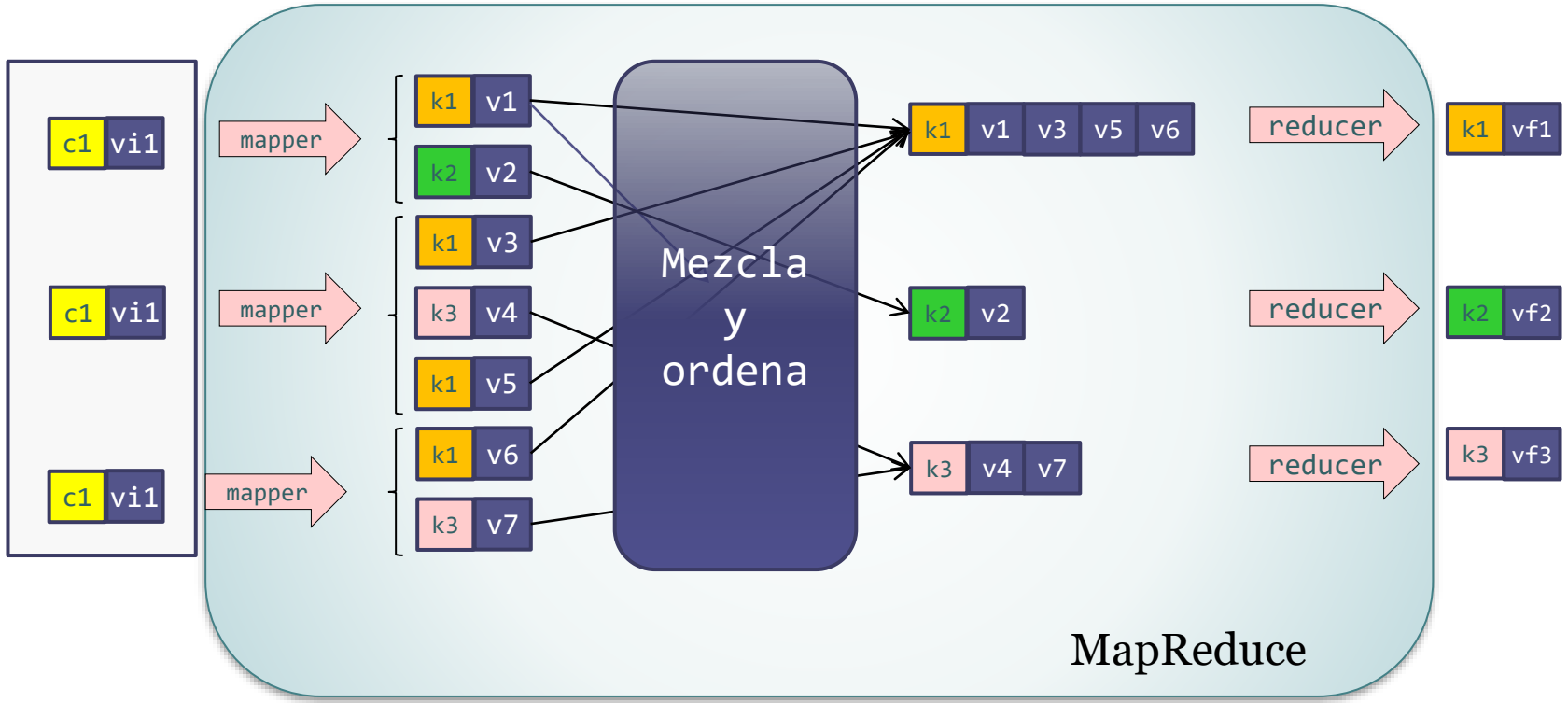
Paso 3 - Reducer

Para cada clave2, toma la lista de valores asociada y los combina en uno solo

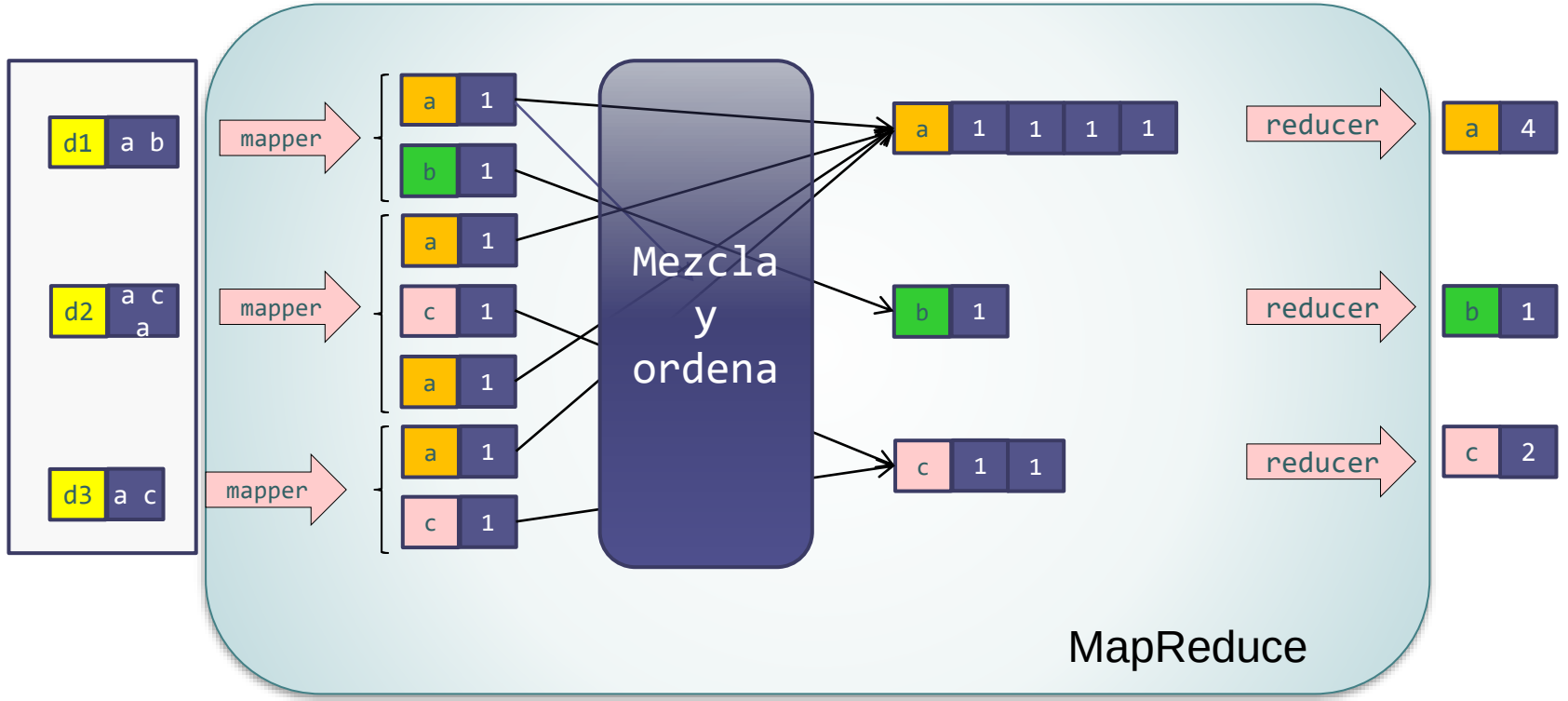
Tipo: `(clave2, [valor2]) → (clave2,valor2)`



MapReduce - Esquema general



MapReduce - Cuenta palabras




```
// devuelve cada palabra con un 1
mapper(d,ps) {
 for each p in ps:
   emit (p, 1)
}
```

```
// suma la lista de números de cada palabra
reducer(p,ns) {
  sum = 0
  for each n in ns { sum += n; }
  emit (p, sum)
}
```

MapReduce - Entorno ejecución

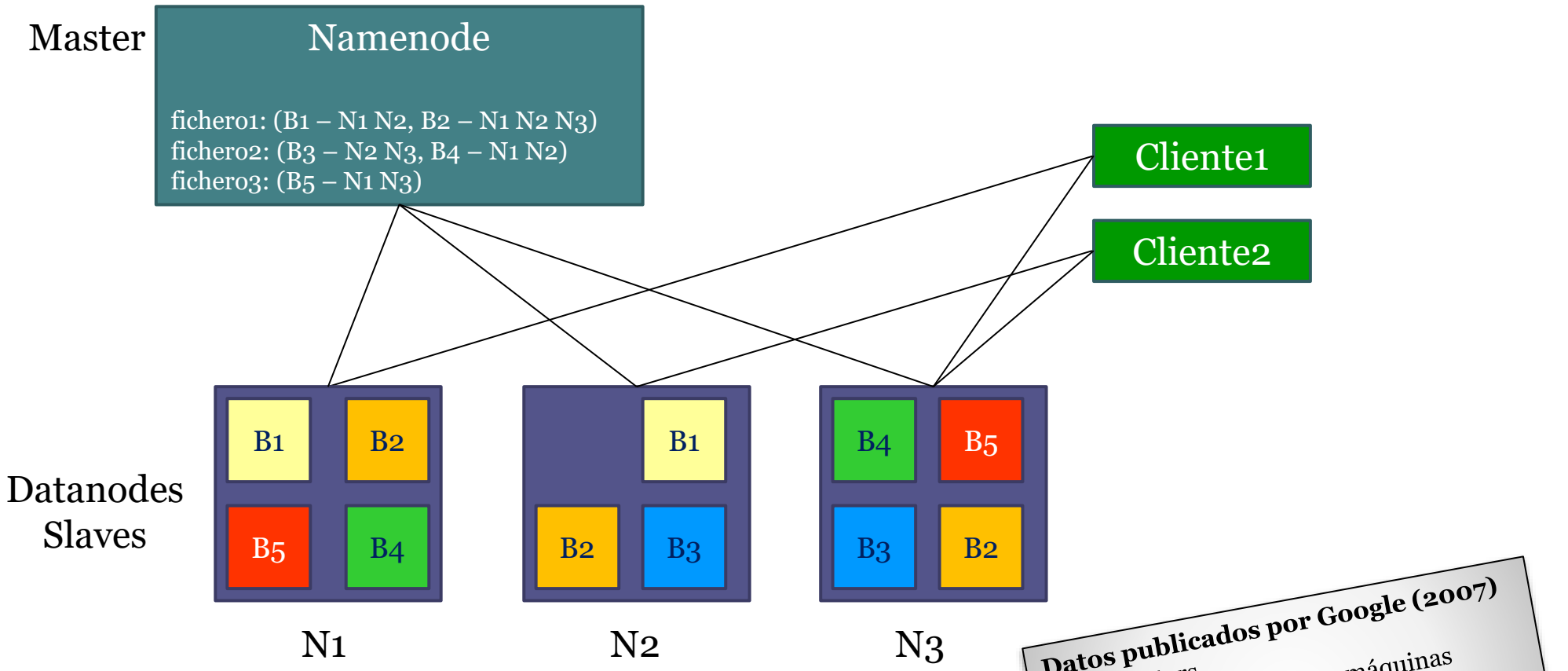
El entorno de ejecución se encarga de

- Planificación: Cada trabajo (*job)* se divide en tareas (*tasks)*
- Co-localización de datos/código
  - Cada nodo computacional contiene sus datos de forma local (no existe un sistema central)
- Sincronización:
  - Tareas *reduce* deben esperar final de fase *map*
- Gestión de errores y fallos
  - Alta tolerancia a fallos de los nodos computacionales

MapReduce - Sistema de ficheros

- Google desarrolló sistema distribuido GFS
- Hadoop creó HDFS
  - Ficheros se dividen en bloques (chunks)
  - 2 tipos de nodos:
    - Namenode (maestro), datanodes (servidores datos)
  - Datanodes almacenan diferentes bloques
    - Replicación de bloques
  - *Namenode* contiene metadatos
    - En qué nodo está cada trozo
    - Comunicación directa entre clientes y datanodes

MapReduce - Sistema de ficheros



Datos publicados por Google (2007)
200+ clusters
Muchos clusters de 1000+ máquinas
Pools de miles de clientes
4+ PB
Tolerancia fallos de HW/SW

MapReduce

Ventajas

- Computaciones distribuidas
  - Troceado de datos de entrada
    - Replicated repository
  - Tolerancia a fallos de nodos
    - Hardware/software heterogéneo
- Procesamiento grandes cantidades de datos
  - Write-once. Read-many

Problemas

- Dependencia Nodo Maestro
- No interactividad
- Conversión datos
  - Adaptar datos de entrada
  - Convertir datos obtenidos

MapReduce: Aplicaciones

Múltiples aplicaciones:

- Google en 2007, 20petabytes al día, en una media de 100mil trabajos mapreduce/día
- El algoritmo PageRank puede implementarse mediante MapReduce
- Casos de éxito:
  - Traducción automática, Similaridad entre ítems, ordenamiento (Hadoop ordena 500GB/59sg (véase: sortbenchmark.org)
- Otras compañías: last.fm, facebook, Yahoo!, twitter, etc.

MapReduce: Implementaciones

Google (interna)
Hadoop (*open source)*
CloudMapReduce (basado en servicios de Amazon)
Aster Data (SQL)
Greenplum (SQL)
Disco (Python/Erlang)
Holumbus (Haskell)
…

MapReduce: Librerías/lenguajes

Hive (Hadoop): lenguaje de consulta inspirado en SQL
Pig (Hadoop): lenguaje específico para definir flujos de datos
Cascading: API para especificar flujos de datos distribuidos
Flume Java (Google)
Dryad (Microsoft)

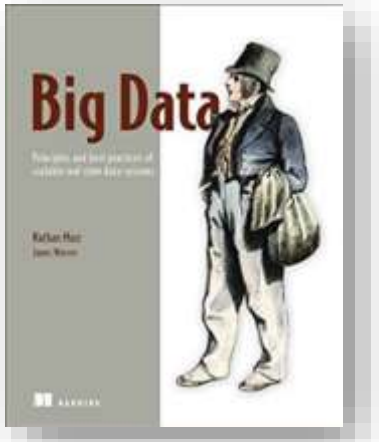
Arquitectura lambda

Afrontar análisis de Big data en tiempo real

Propuesta por Nathan Marz (2011)

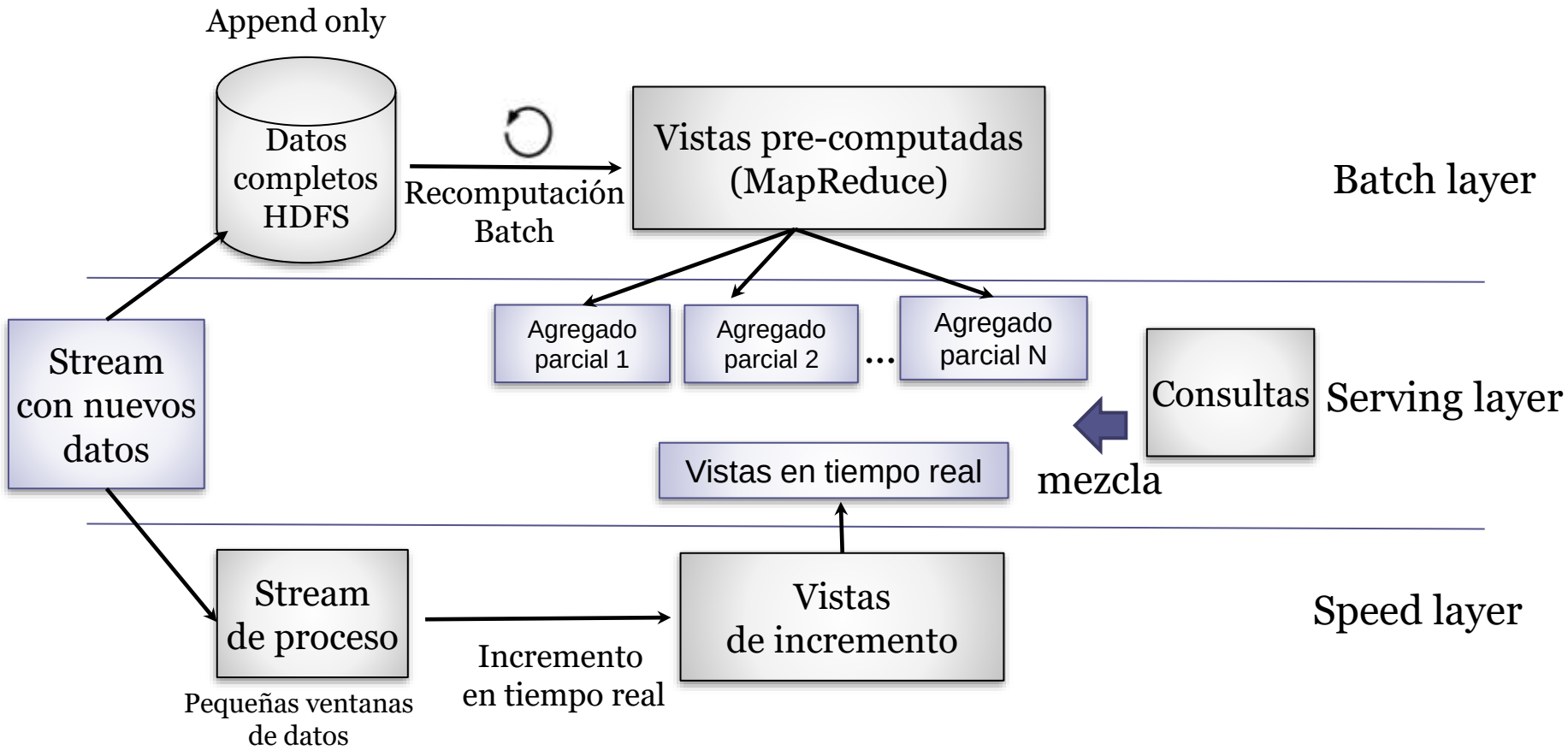
3 capas

- Batch layer: pre-computa todos los datos (mapReduce)
  - Genera vistas agregadas parciales
  - Re-calcula todos los datos cada cierto tiempo
- Speed layer : Tiempo real, ventana de datos
  - Genera vistas en tiempo real rápidas
- Serving layer: gestiona consultas
  - Mezcla las vistas diferentes



Arquitectura lambda

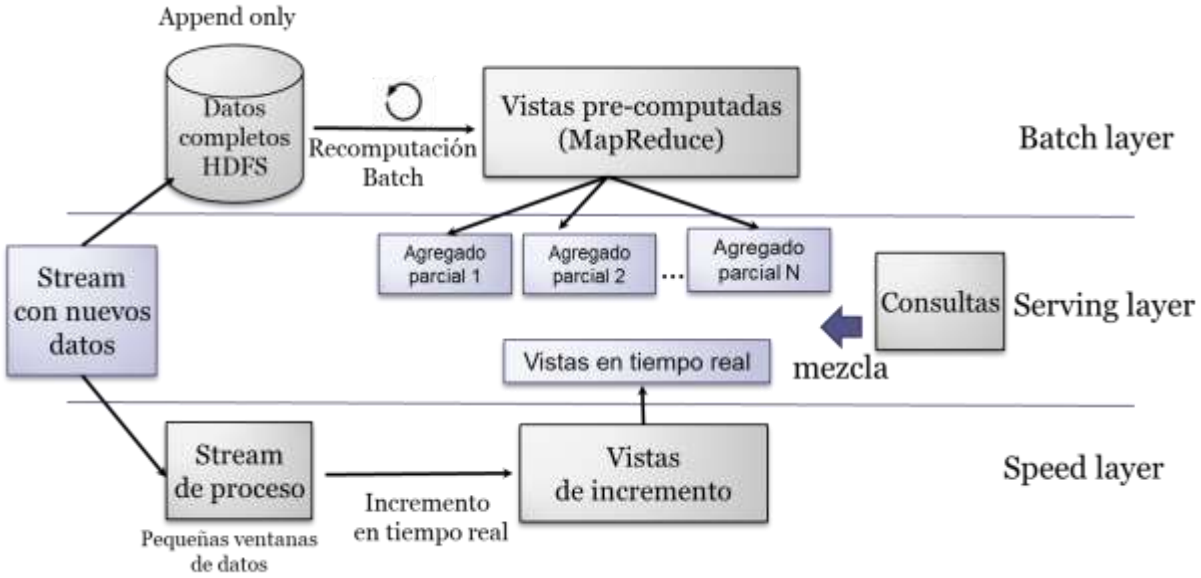
Combina procesamiento Batch con procesamiento en tiempo real



Arquitectura Lambda

Restricciones

- Todos los datos se almacenan en la batch layer
- La *batch layer* precomputa las vistas
- Los resultados de la speed layer podrían no ser exactos
- La Serving layer combina vistas pre-computadas
- Las vistas pueden ser simples bases de datos para consultas



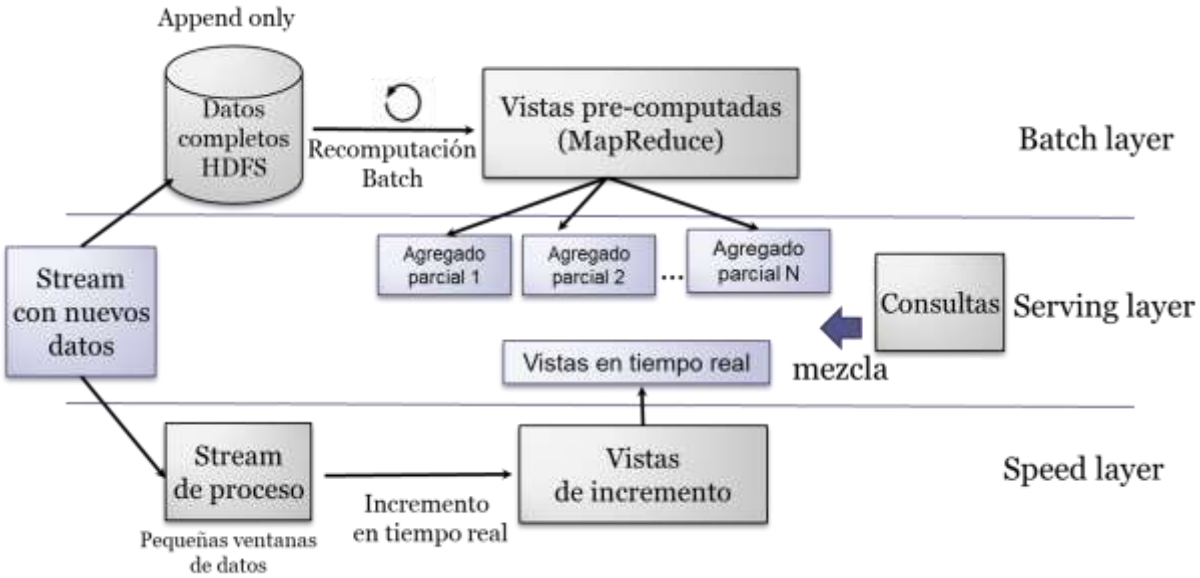
Arquitectura Lambda

Ventajas

- Escalabilidad (Big data)
- Tiempo real
- Desacoplamiento
- Tolerancia a fallos
- Mantiene todos los datos de entrada
  - Se pueden reprocesar

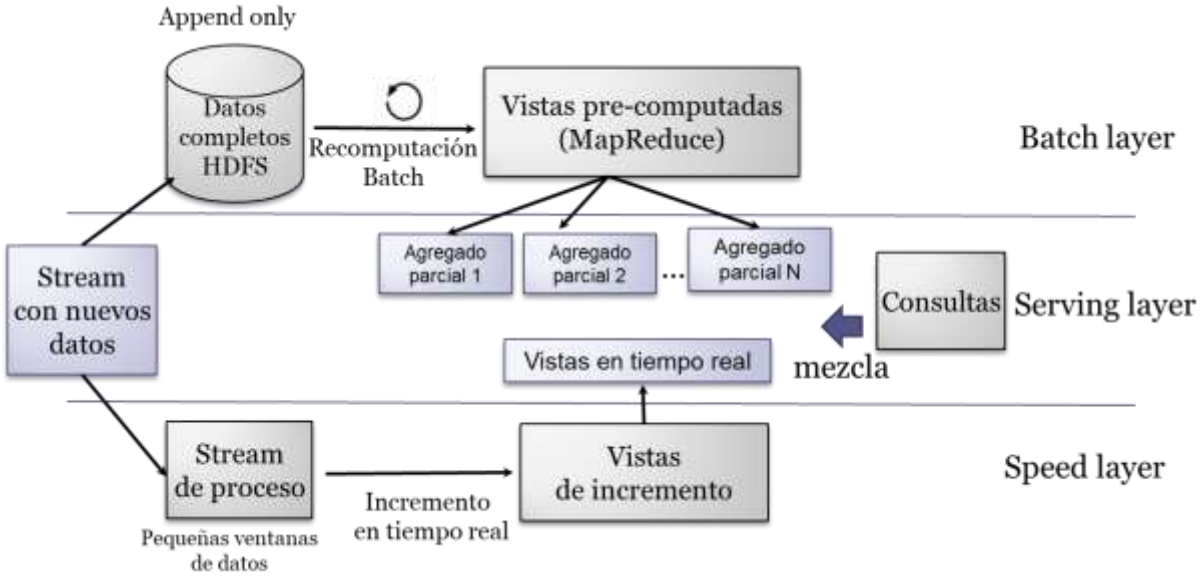
Problemas

- Complejidad inherente
- Las vistas podrían no ser exactas
  - Se podrían perder eventos



Arquitectura Lambda

- Aplicaciones
  - Muchas compañías para analítica de datos
  - Spotify, Alibaba,…
- Librerías
  - Apache Storm
  - Proyecto Netflix Suro

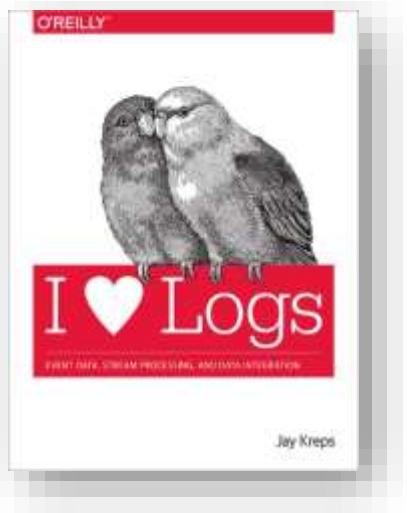


Arquitectura Kappa

Propuesta por Jay Krepps (Apache Kafka, 2013)

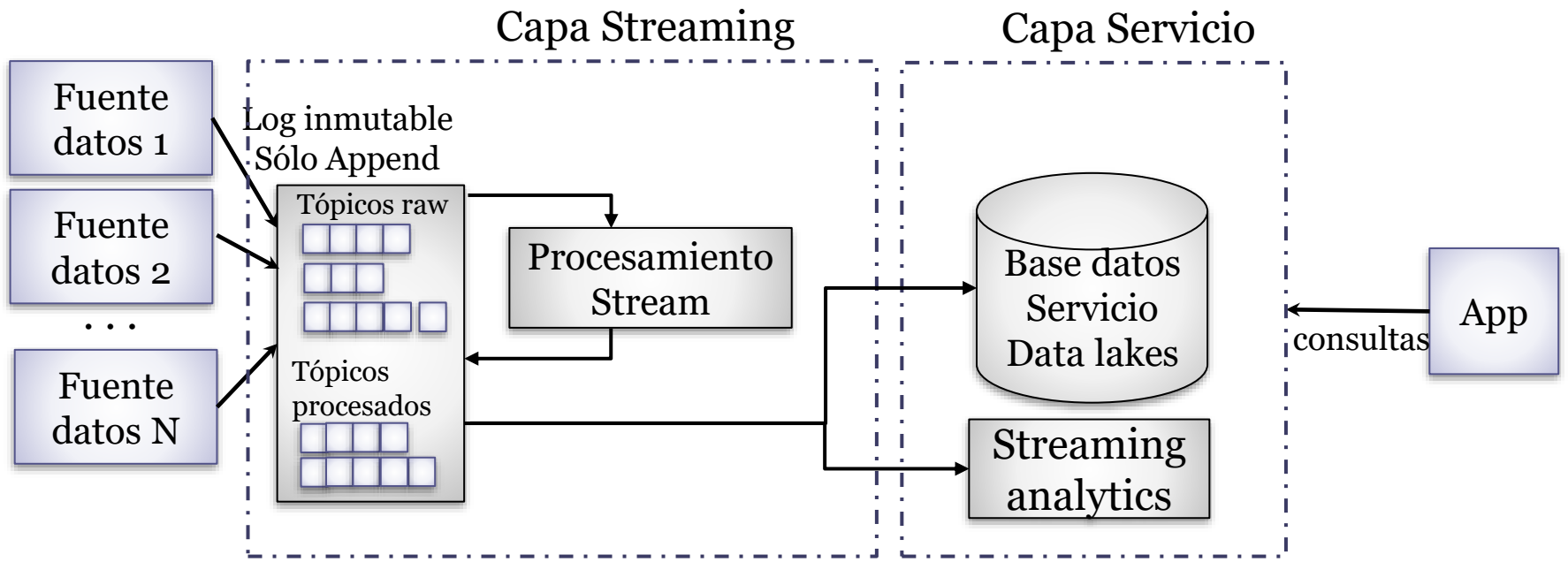
Afrontar Big data & Tiempo real mediante logs

- Simplificar arquitectura Lambda
  - Suprime la batch layer
- Se basa en un log ordenado distribuido
  - Clúster replicado
  - El log puede ser muy grande



Arquitectura Kappa

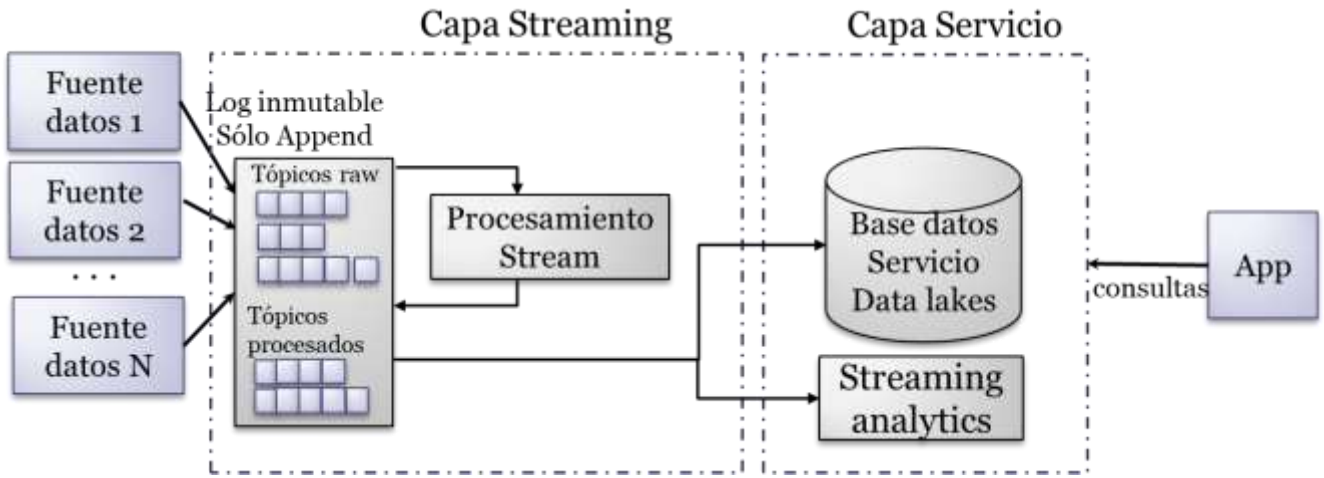
Diagrama



Arquitectura Kappa

Restricciones

- El log de eventos es append-only
- Los eventos en el log son inmutables
- El procesamiento de Streams puede necesitar los eventos en cualquier posición
  - Para gestionar fallos ó hacer recomputaciones



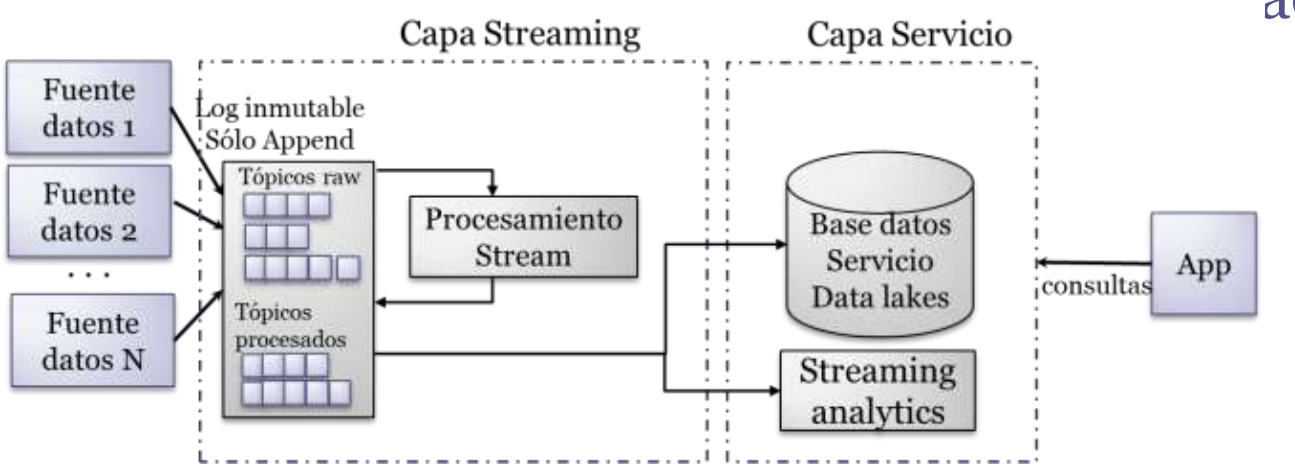
Arquitectura Kappa

Ventajas

- Escalable (big data)
- Tiempo real
- Más simple que arquitectura lambda
  - No hay batch layer

Retos

- Requisitos de espacio
  - Duplicación de log y BD
  - Compactar el log
- Orden de los eventos
- Procesamiento de eventos
  - At least once, At most once (it may be lost), Exactly once



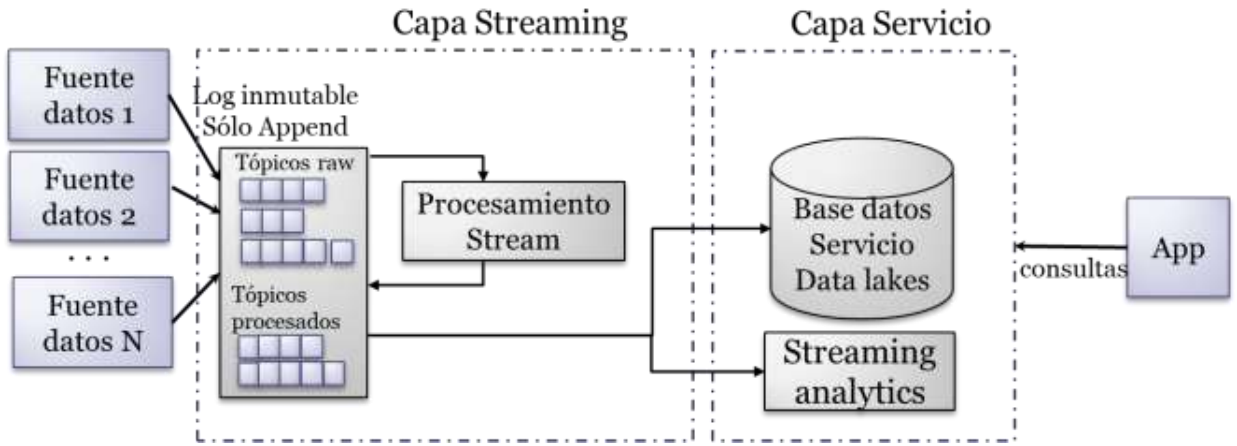
Arquitectura Kappa

Applicaciones

- LinkedIn, Uber, Netflix, VMWare, …

Librerías

- Apache Kafka
- Apache Samza
- Spark Streaming



Fin