



Universidad de Oviedo



Comportamiento

Curso 2020/2021

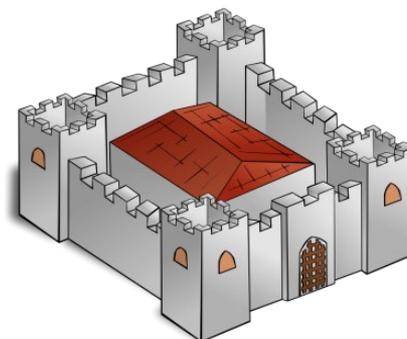
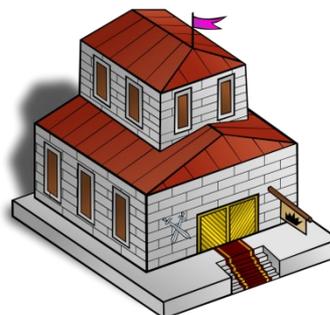


Jose Emilio Labra Gayo

Estilos según comportamiento

Funcionamiento en tiempo de ejecución

Componentes y conectores



1st part.

Estilos básicos y monolitos

Flujos de datos

Secuencial Batch

Pipes & Filters

Pipes & Filters Interfaz Uniforme

Secuencial - Batch

Programas separados son ejecutados en orden
Los datos deben pasarse de un programa al siguiente



Nota

El estilo secuencial (batch) puede considerarse el abuelo de los estilos arquitectónicos



Secuencial - Batch

Elementos:

Programas ejecutables independientes

Restricciones

Encadenar salida de un programa a entrada de otro
Normalmente, un programa debe esperar a que termine la ejecución el programa anterior

Secuencial - Batch

Ventajas

Débil acoplamiento entre componentes

Re-configurabilidad

Depuración

Se puede depurar cada entrada de forma independiente

Problemas

No proporciona interfaz interactivo

Requiere intervención externa

No hay soporte para concurrencia

Baja velocidad (throughput)

Alta latencia

Definiciones:

Throughput: velocidad a la que algo puede procesarse
Ejemplo: n° trabajos/segundo

Latencia: retardo experimentado por un proceso
Ejemplo: 2 segundos

Secuencial - Batch

Problemas

No proporciona interfaz interactivo

Requiere intervención externa

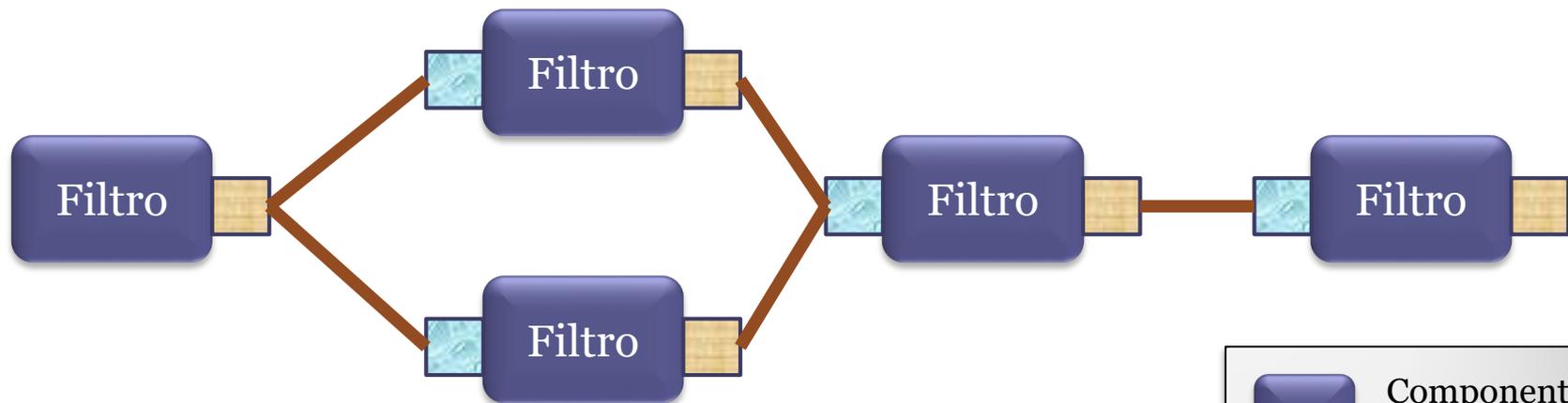
No hay soporte para concurrencia

Baja velocidad (throughput)

Alta latencia

Pipes & Filters

Datos fluyen a través de tuberías (pipes) y son procesados mediante filtros



Pipes & Filters

Elementos

Filtro: componente que transforma los datos.

Los filtros pueden ejecutarse concurrentemente.

Tipos de filtros

Fuentes de datos (entrada al sistema)

Flujo

Sumideros (sinks) (salida del sistema)

Tubería (Pipe): Lleva datos de la salida de un filtro a la entrada de otro filtro

Propiedades: tamaño de búffer, formato de datos, protocolo de interacción

Pipes & Filters

Restricciones

Tuberías conectan salidas de un filtro a entrada de otro
Los filtros deben estar de acuerdo sobre los formatos que admiten

Pipes & Filters

Ventajas

Comprensión global sistema
Comportamiento total = suma
comportamiento de cada
filtro

Reconfiguración:

Filtros pueden recombinarse

Evolución y extensibilidad:

Crear/añadir nuevos filtros

Se pueden sustituir filtros
viejos por nuevos

Testabilidad

Verificación independiente de
cada filtro

Rendimiento

Permite ejecución
concurrente entre filtros

Retos

Posibles retardos si hay
tuberías largas

Difícil pasar estructuras de
datos complejas

No interactividad

Un filtro no puede interactuar
con el entorno

Backpressure

Consumidores que reciben más
cantidad de datos de la que
pueden procesar

Pipes & Filters

Aplicaciones

Unix

`who | wc -l`

Yahoo Pipes

Java Streams

Flow based programming

https://en.wikipedia.org/wiki/Flow-based_programming

Stream programming

Pipes & Filters - interfaz uniforme

Variación de *Pipes & Filters* en la que los filtros tienen la misma interfaz

Elementos

Los mismos que en Pipes & Filters

Restricciones

Los filtros deben tener una interfaz uniforme

Pipes & Filters - interfaz uniforme

Ventajas:

- Facilita desarrollo independiente de filtros

 - Más fácil porque el interfaz es conocido

- Reconfigurabilidad

- Facilita la comprensión del sistema

Problemas:

- Empeora rendimiento si los datos deben transformarse desde su representación original

 - Marshalling

Pipes & Filters - interfaz uniforme

Ejemplos:

Sistema operativo Unix

Programas con una entrada (*stdin*) y dos salidas (*stdout* y *stderr*)

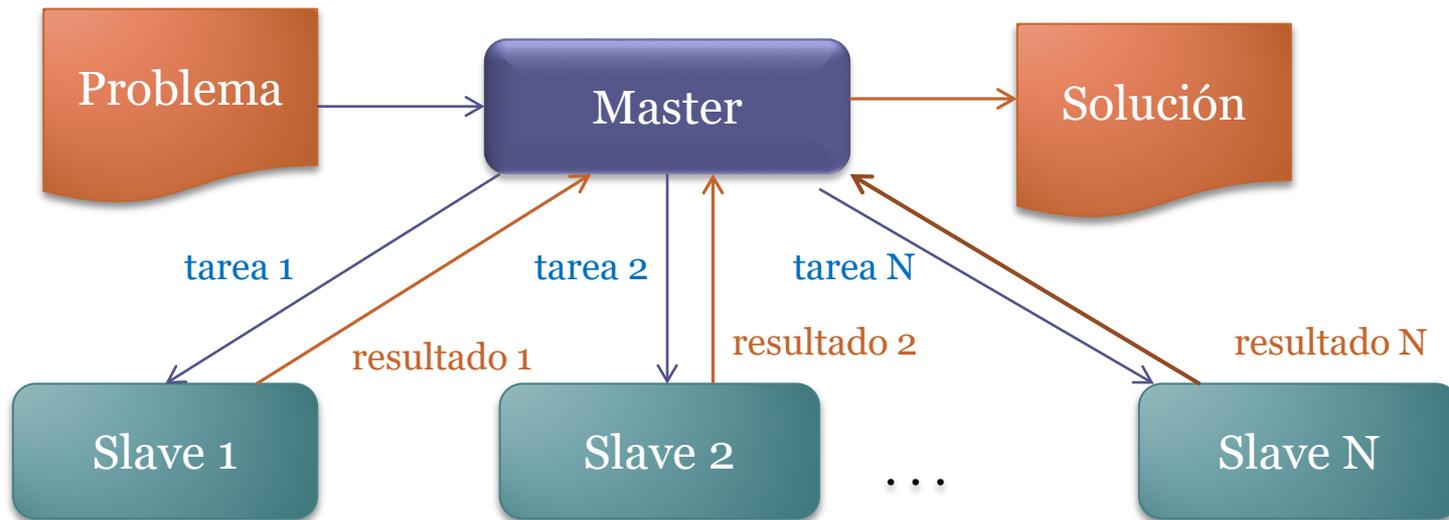
Arquitectura de la Web: REST

Organización del trabajo

Master-Slave

Master-Slave

Maestro divide el trabajo en subtareas
Asigna cada subtarea a diferentes nodos
El resultado computacional se obtiene a partir de los resultados de los nodos esclavos



Master-Slave

Elementos

Master: Se encarga de coordinar la ejecución

Slave: realiza una tarea y devuelve un resultado

Restricciones

Los *slave* se encargan únicamente de realizar la computación

El control es realizado desde el nodo *Master*

Master-Slave

Ventajas

Computación paralela

Tolerancia a fallos

Problemas

Dificultad de coordinación entre nodos *slave*

Dependencia de nodo *Master*

Dependencia de configuración física

Master-Slave

Aplicaciones:

Sistemas de control de procesos

Sistemas empotrados

Sistemas tolerantes a fallos

Sistemas de búsqueda de soluciones precisas

Sistemas interactivos

MVC: Modelo - vista - controlador

Variaciones de MVC

PAC: Presentación - Abstracción - Control

DCI : Datos - Contexto - Interacción

MVC

MVC: Modelo - Vista - Controlador

Trygve Reenskaug, finales de los 70

Solución para GUI

Incorpora un controlador para separar el modelo de la vista que se ofrece al usuario

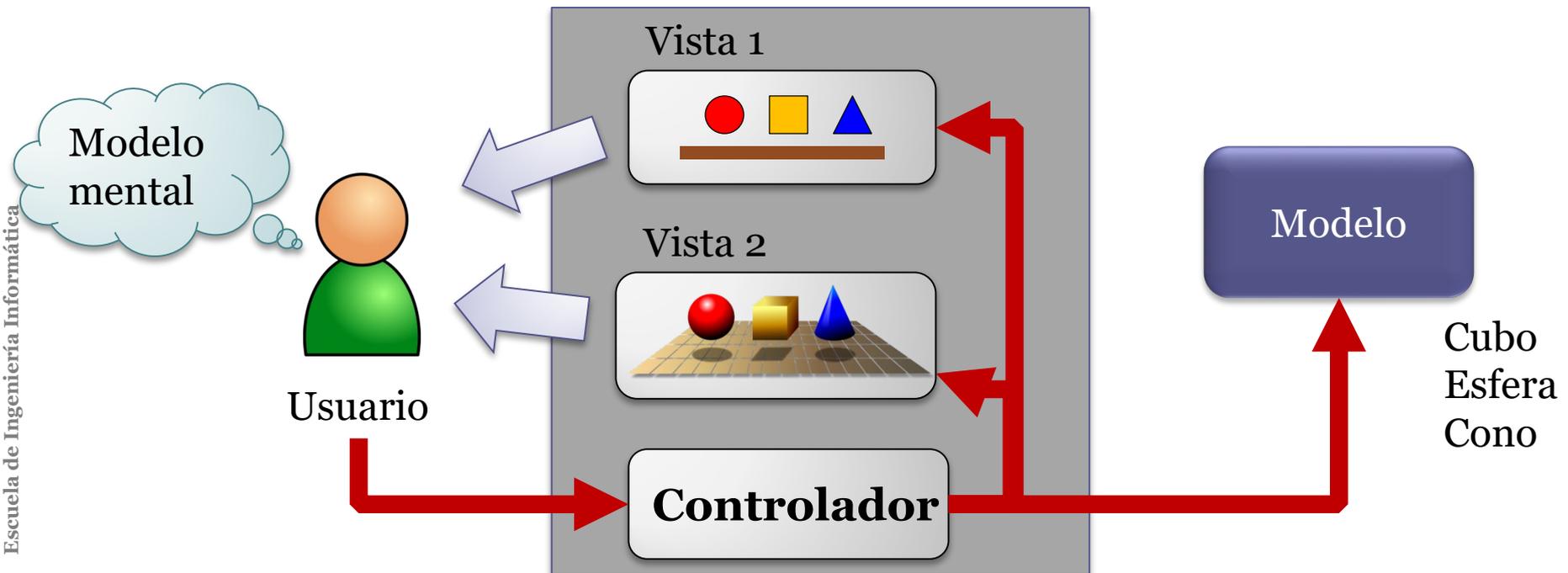
El usuario trabaja con un "modelo mental" del modelo real, que se ofrece a través de vistas

MVC

Modelo: Lógica de negocio y estado

Vista: Muestra datos al usuario

Controlador: Coordina interacción, vistas y modelo



MVC

Elementos

Modelo

Representa lógica de negocio y estado

Enlace con almacén de datos y actualización

Vista

Muestra contenidos de un modelo

Controlador

Recibe interacciones del usuario con la vista

Coordina acciones a realizar por modelo

Creación/coordinación de vistas

MVC

Restricciones

El controlador se encarga de procesar los eventos del usuario

La vista se encarga únicamente de mostrar valores del modelo

Modelo es independiente de controladores/vistas

MVC

Ventajas

- Múltiples vistas del mismo modelo
- Sincronización de vistas
- Separación de incumbencias
 - Interacción (controlador), funcionalidad (modelo)
- Facilidad para crear nuevas vistas y controladores
 - Intercambiar *look & feel*
- Potencial para creación de marcos genéricos

Problemas

- Mayor complejidad en desarrollo de GUIs
- Acoplación entre controladores y vistas
- Controlador/Vistas dependen del interfaz del modelo
- Dificultades con herramientas GUI

MVC

Aplicaciones

Muchos marcos de aplicación Web siguen MVC

Ruby on Rails, Spring MVC, Play, etc.

Diferencia

Push: el controlador envía órdenes a la vista

Ruby on Rails, Struts1

Pull: el controlador recibe órdenes de la vista

Play! Framework, Struts2

Variaciones de MVC

PAC

Model-View-Presenter

Model View ViewModel

Model View Update

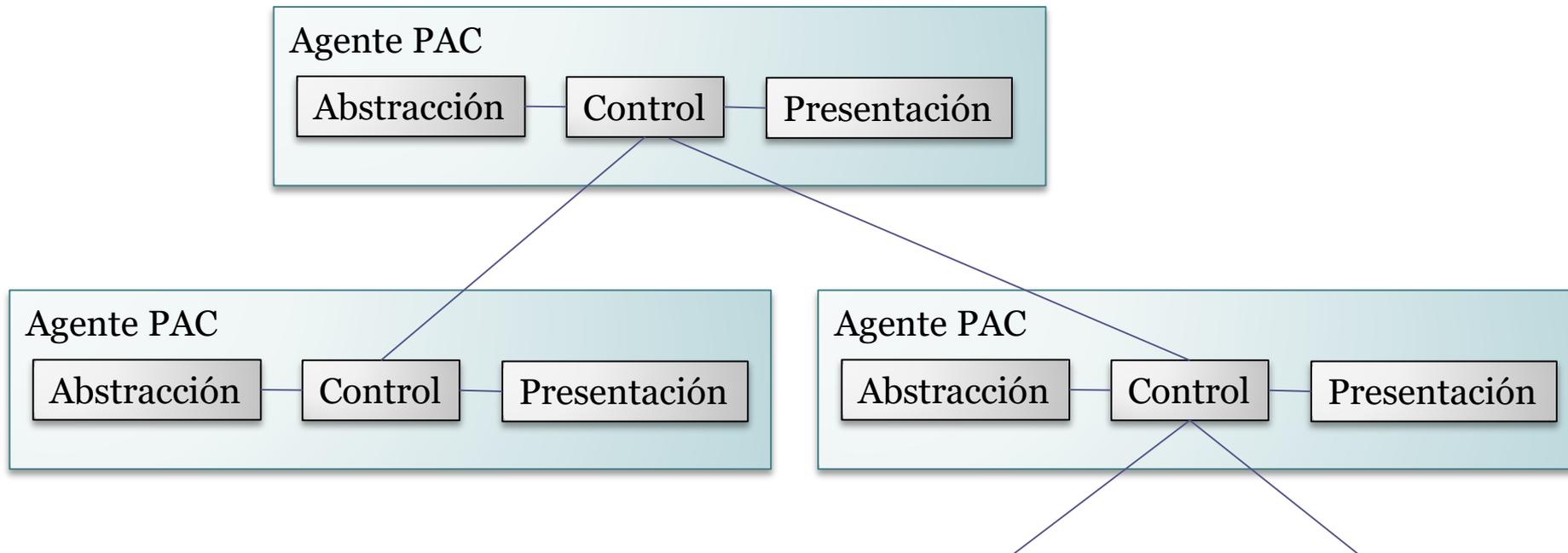
...

PAC

PAC: Presentación-Abstracción-Control

Jerarquía de agentes

Cada agente contiene 3 componentes



PAC

Elementos

Agentes con

Presentación: aspecto de visualización

Abstracción: modelo de datos de un agente

Control: conecta los componentes anteriores y permite la comunicación entre agentes

Relación jerárquica entre agentes

Restricciones

Cada agente se encarga de un aspecto de la funcionalidad

En cada agente no hay comunicación directa entre Abstracción y Presentación

Comunicación a través de componente de control

PAC

Ventajas

Separación de responsabilidades

Soporte para cambios y extensiones

modificar un agente sin modificar el resto

Multitarea

Los agentes pueden ejecutarse en paralelo

Problemas

Complejidad del sistema

Demasiados agentes pueden generar una estructura compleja y difícil de mantener

Complejidad del componente de control

Componentes de control gestionan comunicación
Su calidad es fundamental para la calidad del sistema

Rendimiento

Sobrecarga de comunicación entre agentes

PAC

Aplicaciones

Sistema de monitorización de redes

Robots móviles

Relaciones

Relacionado con MVC

En MVC el componente de Presentación se separa en Vista y Controlador

En MVC no hay componentes de control ni jerarquía de agentes.

Redescubierto y rebautizado como MVC jerárquico

Repositorio

Datos compartidos

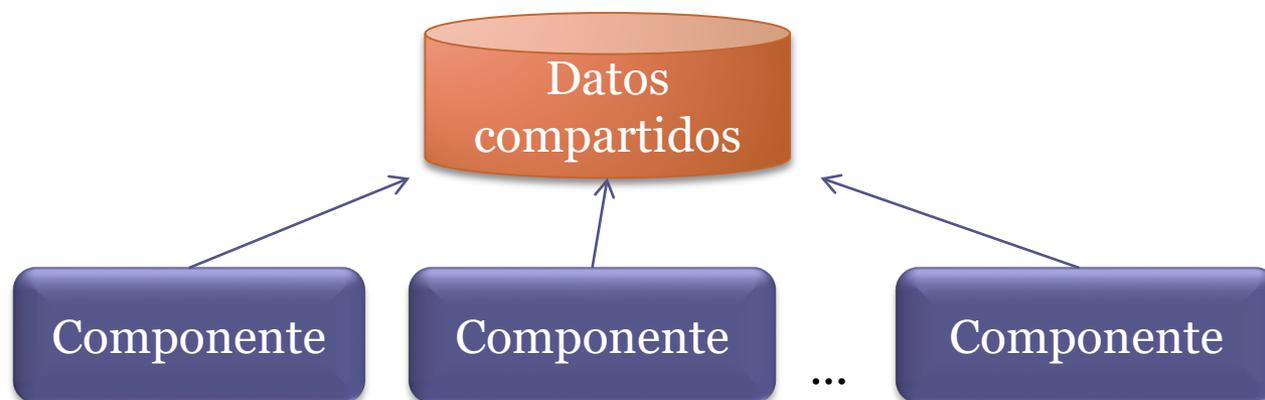
Blackboard

Basados en reglas

Datos compartidos

Varios componentes independientes acceden al mismo estado

Aplicaciones basadas en un modelo centralizado



Datos compartidos

Elementos

Almacén de datos

Base de datos o repositorio centralizado

Componentes

Procesadores que acceden a la memoria compartida

Datos compartidos

Restricciones

Componentes actúan sobre el estado global

Los componentes no se comunican entre sí

Sólo a través del estado compartido

El estado garantiza la estabilidad de los datos

Datos compartidos

Ventajas

Componentes
independientes

No necesitan conocer
existencia de otros
componentes

Facilita comunicación entre
componentes

Consistencia de datos

Estado global centralizado
Backup único de todo el
sistema

Problemas

Punto de fallo único

Fallo del almacén puede
comprometer todo el
sistema

Distribución del almacén
puede ser costosa

Posible cuello de botella

Ineficiencia en
comunicación

Sincronización en acceso
a memoria compartida

Datos compartidos

Aplicaciones

Gran cantidad de sistemas utilizan este esquema

Algunas variantes

Este patrón se conoce también como:

Shared Memory, Repository, Shared data, etc.

Blackboard

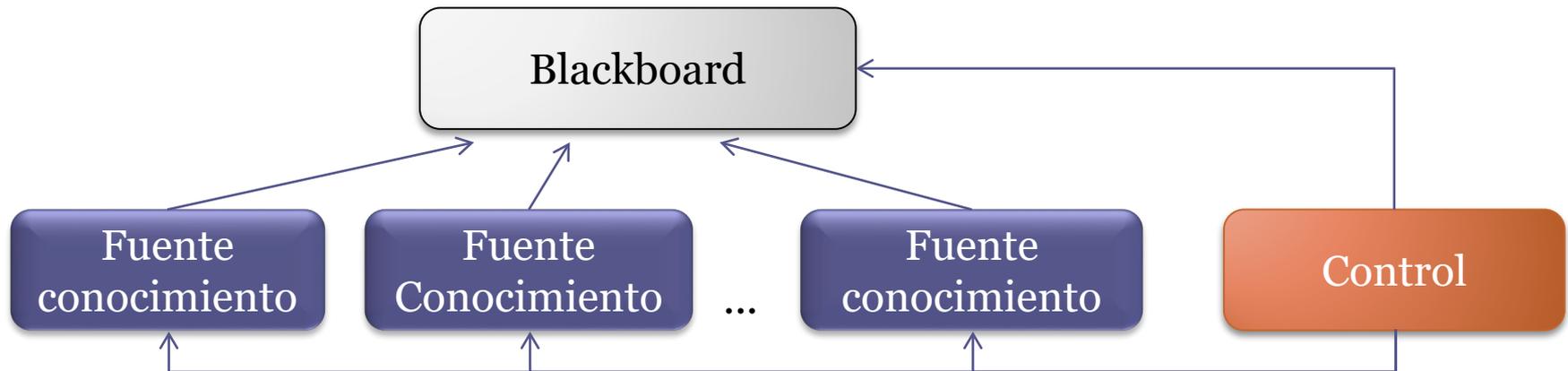
Sistemas basados en reglas

Blackboard

Problemas complejos de difícil solución

Se dividen en *fuentes de conocimiento* que resuelven partes del problema

Cada fuente de conocimiento agrega soluciones parciales en el *blackboard*



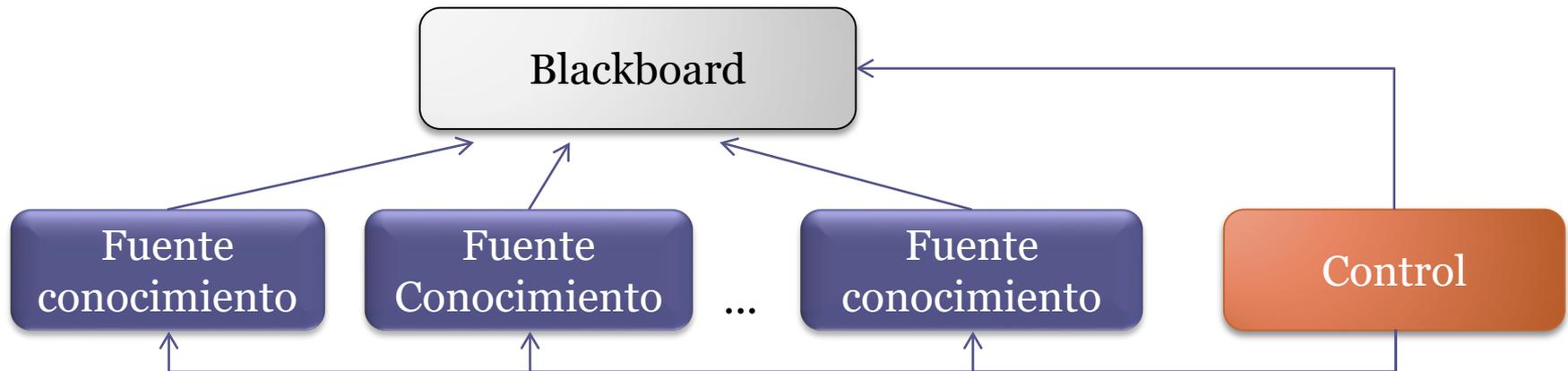
Blackboard

Elementos

Blackboard: Almacén de datos central

Fuente de conocimiento: resuelve una parte del problema y va añadiendo los resultados parciales

Control: Organiza tareas y chequea estado del trabajo



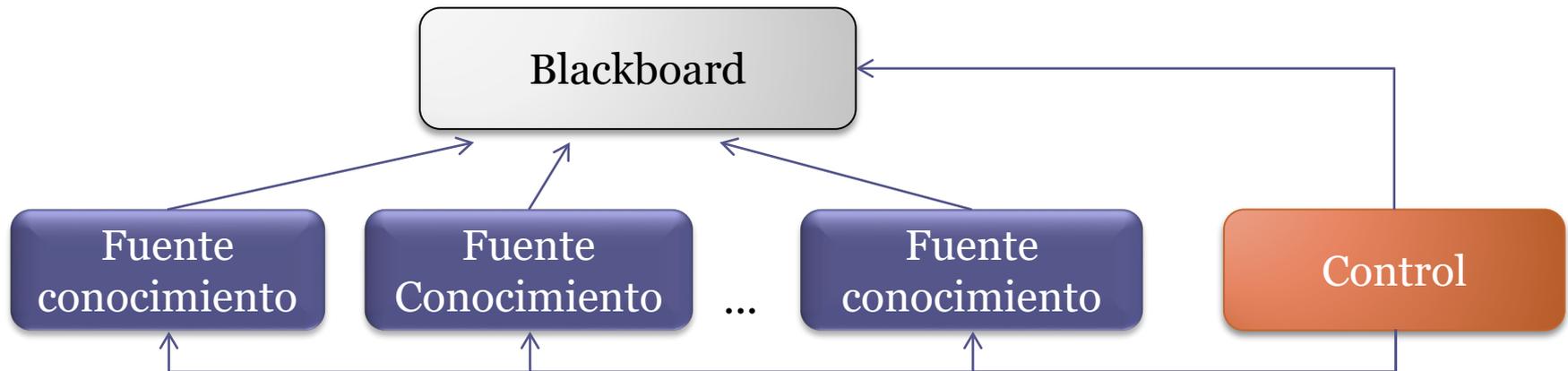
Blackboard

Restricciones

El problema se descompone en partes

Cada fuente de conocimiento sólo resuelve una **parte** del problema

El *blackboard* contiene soluciones parciales que van mejorándose



Blackboard

Ventajas

Experimentación

Aplicable para
problemas abiertos

Facilita cambio de
estrategias

Reusabilidad

Fuentes de
conocimiento
reutilizables

Tolerancia a fallos

Problemas

Depuración

No hay garantía de encontrar
solución adecuada

Dificultad para establecer
estrategia central de control

Rendimiento

Puede ser necesario rechazar
hipótesis incorrectas

Alto coste de desarrollo

Implementación del
paralelismo

Necesidad de sincronizar
acceso al *blackboard*

Blackboard

Aplicaciones

Sistemas de reconocimiento del habla

HEARSAY-II

Reconocimiento de patrones

Predicción atmosférica

Juegos

Análisis de estructura molecular

Crystalis

Sistemas basados en reglas

Variante de memoria compartida

Memoria compartida = Base de conocimiento

Contiene reglas y hechos



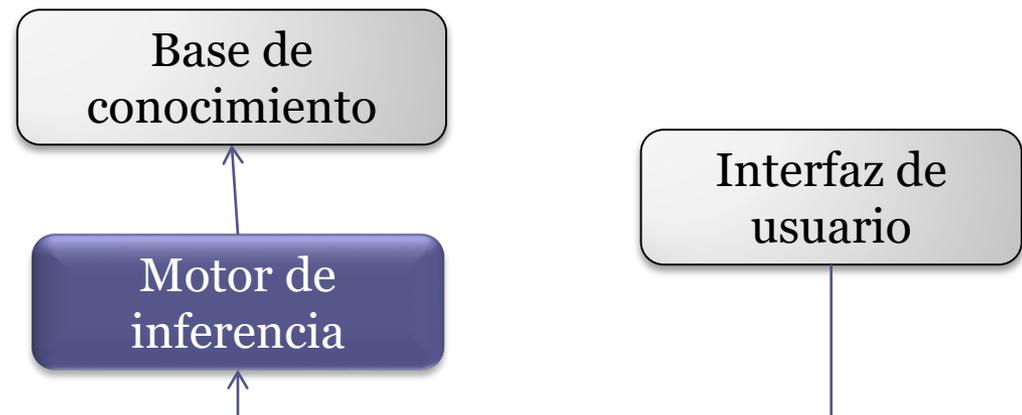
Sistemas basados en reglas

Elementos:

Base de conocimiento: Conjunto de hechos y reglas sobre un determinado dominio

Interfaz de usuario: Accede a la base de conocimiento para consultar/modificar

Motor de inferencia: Sistema encargado de responder consultas a partir de los datos y la base de conocimiento



Sistemas basados en reglas

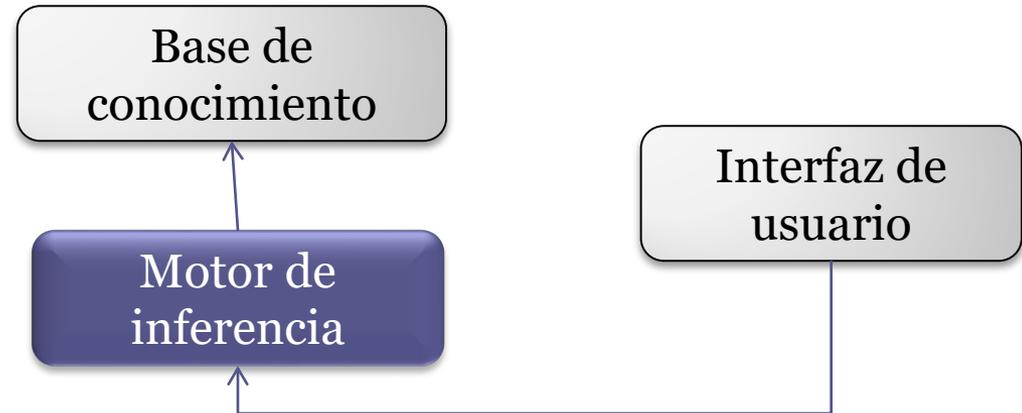
Restricciones:

Base de conocimiento declarativa

Se basa en reglas del tipo:

IF *antecedentes* THEN *consecuente*

Expresividad limitada respecto lenguajes imperativos



Sistemas basados en reglas

Ventajas

Mantenibilidad

- Solución declarativa
- Puede ser gestionada por expertos del dominio

Separación de responsabilidades

- Algoritmo
- Conocimiento del dominio

Reutilización

Problemas

Depuración

Rendimiento

- Sistema de inferencia
- Creación y mantenimiento de las reglas
- Actualización de las reglas en tiempo de ejecución
- Aprendizaje de nuevas reglas
- Introspección

Sistemas basados en reglas

Aplicaciones

Sistemas expertos

Sistemas de producción

Librerías de reglas en Java

JRules, Drools, JESS

Lenguajes basados en reglas

Prolog

BRMS - Business rules management systems

Invocación

Call-return

Cliente-Servidor

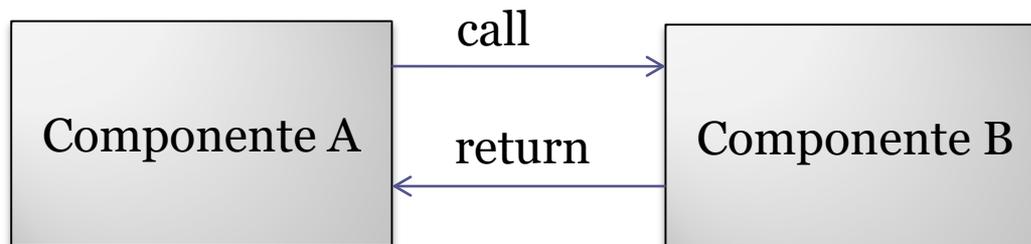
Arquitecturas basadas en eventos

Publish-Subscribe

Modelos de Actores

Call-return

Un componente realiza una llamada a otro componente y espera a recibir la respuesta



Call-return

Elementos

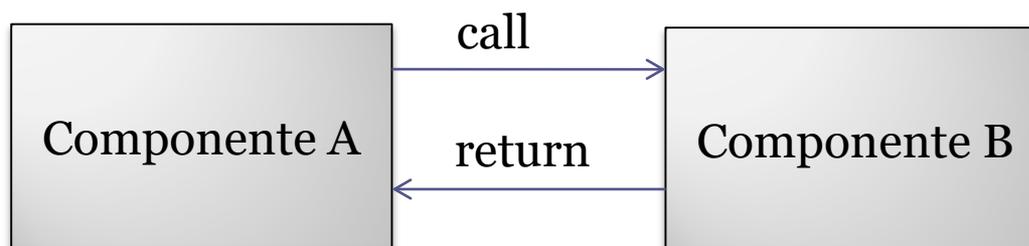
Componente que realiza la llamada

Componente que devuelve la respuesta

Restricciones

Comunicación síncrona:

Componente que realiza llamada queda esperando la respuesta.



Call-return

Ventajas

- Sencillo de implementar

Problemas

- Problemas para ejecución concurrente

 - Componente queda bloqueado esperando respuesta

 - Puede estar ocupando recursos generando bloqueos

- Entornos distribuidos

 - Poco aprovechamiento de capacidades computacionales

Cliente-Servidor

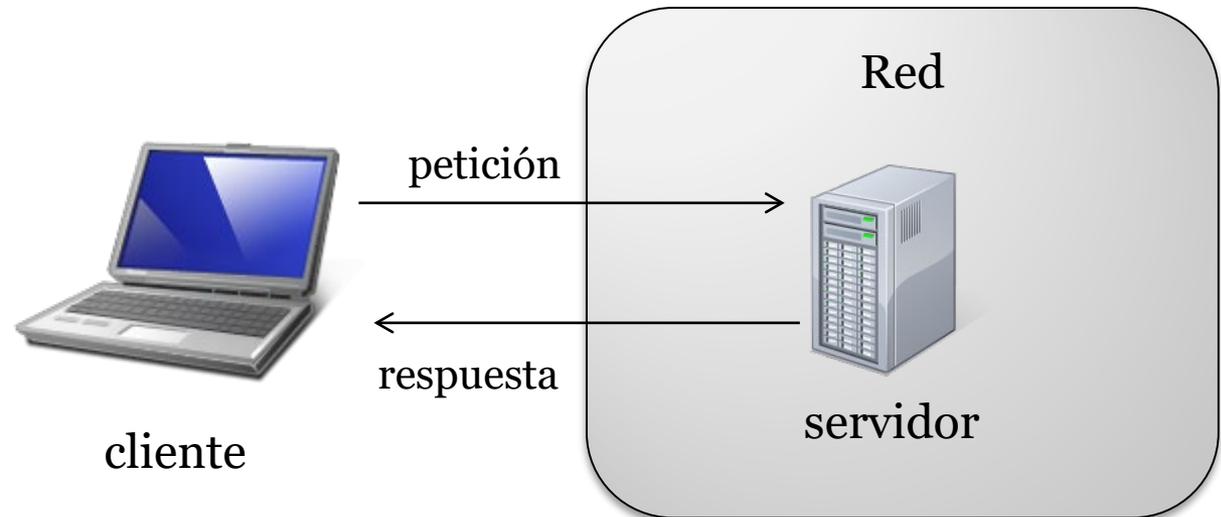
Variación de sistemas en capas

2 capas separadas físicamente (2-tier)

Funcionalidad separada en varios servidores

Clientes se conectan a los servicios

Interfaz Petición/respuesta



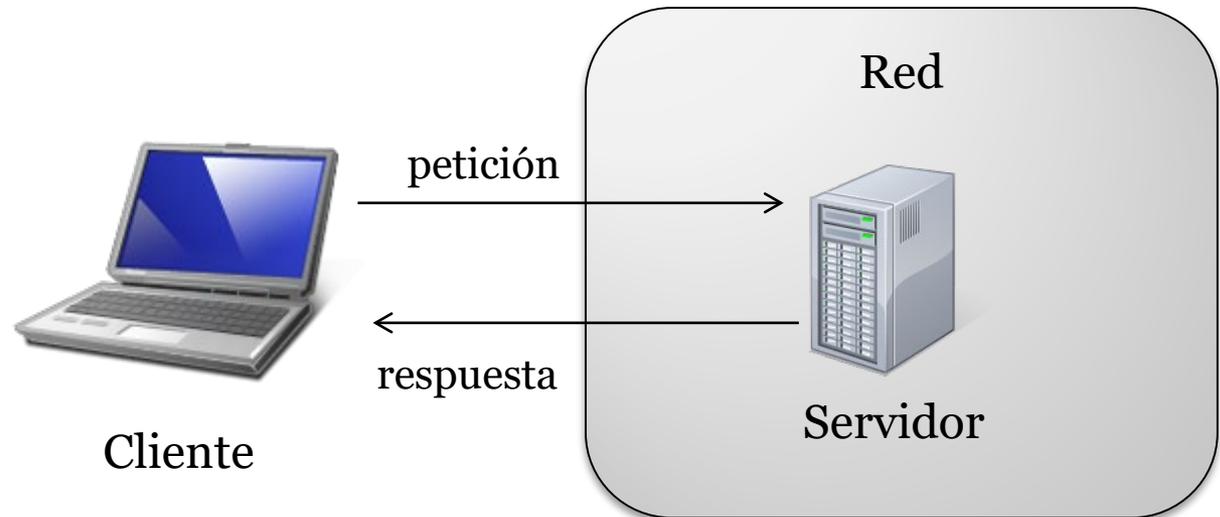
Cliente-Servidor

Elementos

Servidor: ofrece servicios a través de un protocolo petición/respuesta

Cliente: realiza peticiones y procesa las respuestas

Protocolo de red: gestión de comunicación entre clientes y servidores



Cliente-Servidor

Restricciones

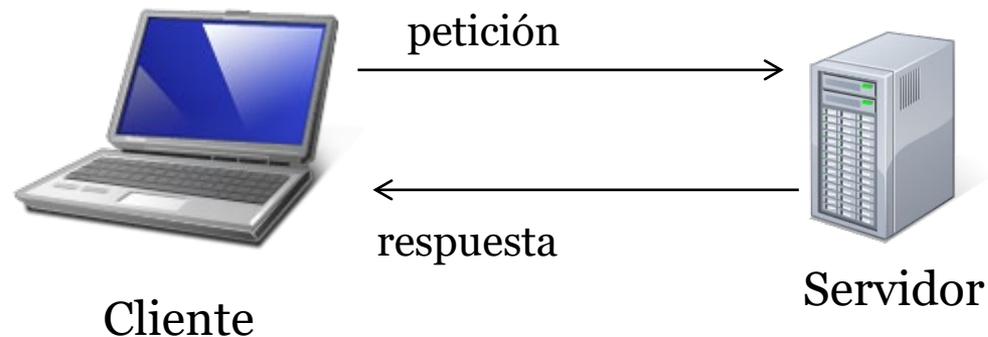
Cientes se comunican con servidores

No al revés

Cientes son independientes de otros clientes

Los servidores no conocen a los clientes

Protocolo de red ofrece garantías de comunicación



Cliente-Servidor

Ventajas

Servidores pueden estar distribuidos

Separación de funcionalidad cliente/servidor

Desarrollo independiente

Escalabilidad

Funcionalidad general disponible para todos los clientes

Aunque no todos los servidores deben ofrecer toda la funcionalidad

Problemas

Cada servidor puede ser un punto de fallo

Ataques a un servidor

Rendimiento impredecible

Dependencia de la red y del sistema

Seguridad

Problemas si los servidores pertenecen a otras organizaciones

Cómo garantizar calidad de servicio

Cliente-Servidor

Variantes

Sin estado

Servidor replicado

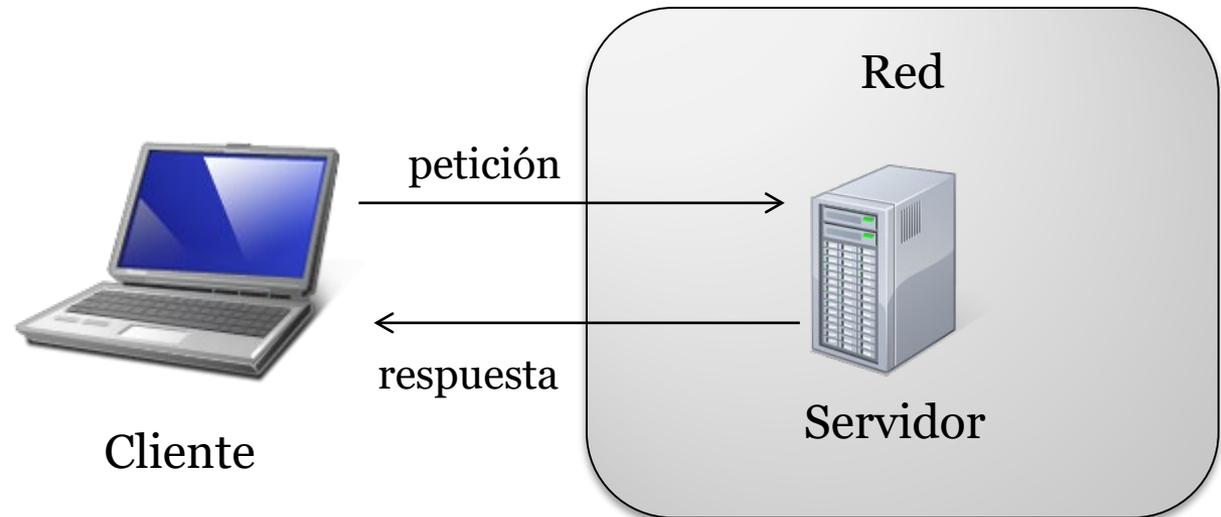
Con caché

Cliente-Servidor sin estado

Restricción:

El servidor no almacena información sobre los clientes

Ante la misma petición responde la misma respuesta



Cliente-Servidor sin estado

Ventajas

- Escalabilidad

Problemas

- Gestión del estado de la aplicación

 - Cliente debe recordar peticiones

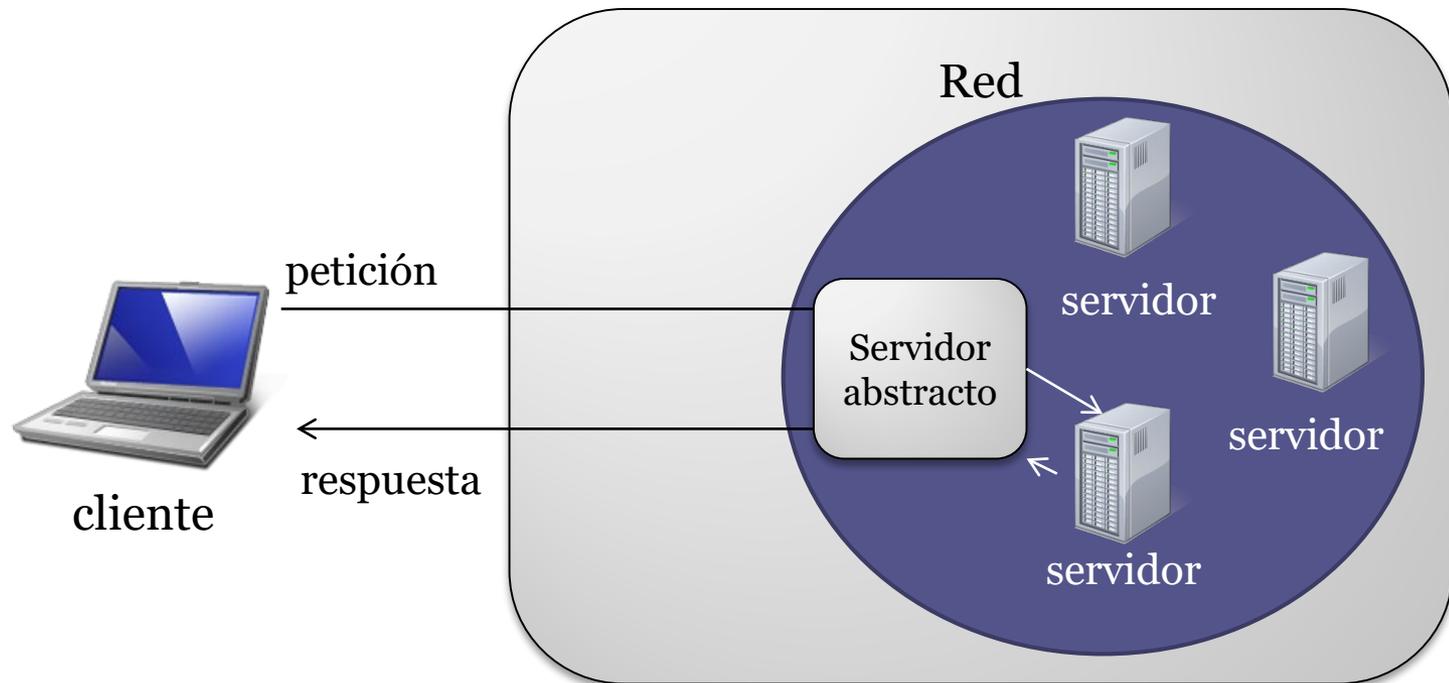
 - Estrategias mantener información entre peticiones

Servidor Replicado

Restricción

Varios servidores ofrecen el mismo servicio

Ofrecer al cliente la ilusión de que solamente hay un servidor



Servidor Replicado

Ventajas

Mejora tiempos de respuesta

Menor latencia

Tolerancia a fallos

Problemas

Mantenimiento de consistencia

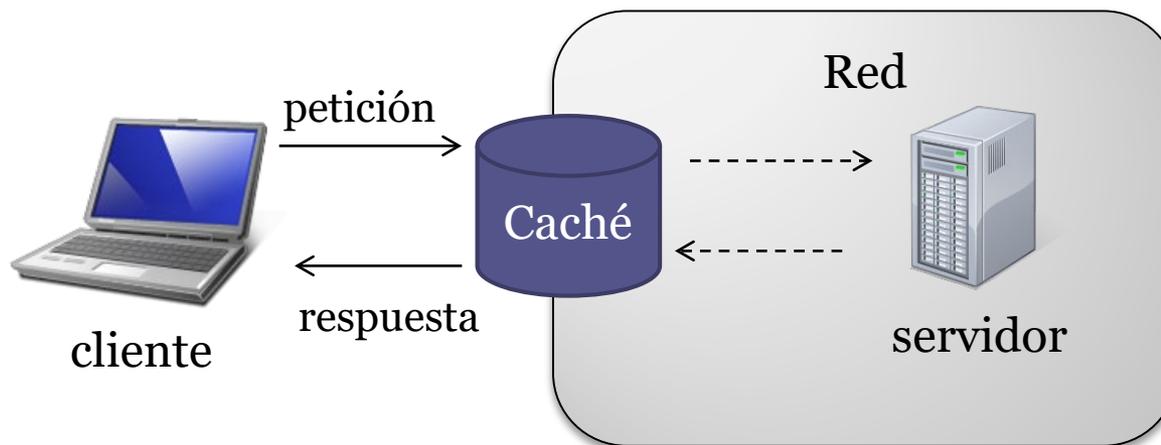
Sincronización

Cliente-servidor con caché

Caché = mediador entre cliente/servidor

Almacena copias de respuestas anteriores a peticiones del servidor

Cuando se repite la petición, se devuelve la respuesta sin necesidad de consultar el servidor



Cliente-servidor con caché

Elementos:

Añade nodos intermedios con caché

Restricciones

Algunas peticiones se resuelven en el nodo caché

El nodo caché contiene política de gestión de respuestas

Tiempo de expiración

Cliente-servidor con caché

Ventajas:

Menor carga en la red

Muchas peticiones repetidas se almacenan en caché

Menor tiempo de respuesta

Respuestas de caché llegan antes

Problemas

Complejidad de configuración

Se requiere política de expiración

No apropiado en ciertos dominios

Si se requiere fidelidad de respuestas

Ej. sistemas en tiempo real

Arquitectura basada en eventos

EDA (Event-Driven-Architecture)



Basados en eventos

Elementos:

Evento:

Algo que ha ocurrido (\neq petición)

Productor de eventos

Generador de eventos (sensores, sistemas, ...)

Consumidor de eventos

BD, aplicaciones, cuadros de mando, ...

Procesador de eventos

Canal de transmisión

Procesador que filtra y transforma eventos



Basados en eventos

Restricciones:

Comunicación asíncrona

Productores generan eventos en cualquier momento
A los consumidores les llegan eventos en cualquier momento

Relación uno-a-muchos

Un evento puede ser enviado a varios consumidores



Basados en eventos

Ventajas

Desacoplamiento

Productor de eventos no depende del consumidor, ni viceversa.

Atemporalidad

Los eventos se publican sin necesidad de esperar por la finalización de un ciclo

Asincronicidad

Para publicar un evento no es necesario esperar a terminar de procesar otro evento

Problemas

Solución no secuencial

Posible pérdida de control

Consistencia del sistema

Dificultad de depuración



Basados en eventos

Aplicaciones

Redes de procesamiento de eventos

Event-Stream-Processing (ESP)

Complex-event-processing

Variaciones

Publish-subscribe

Modelos de actores

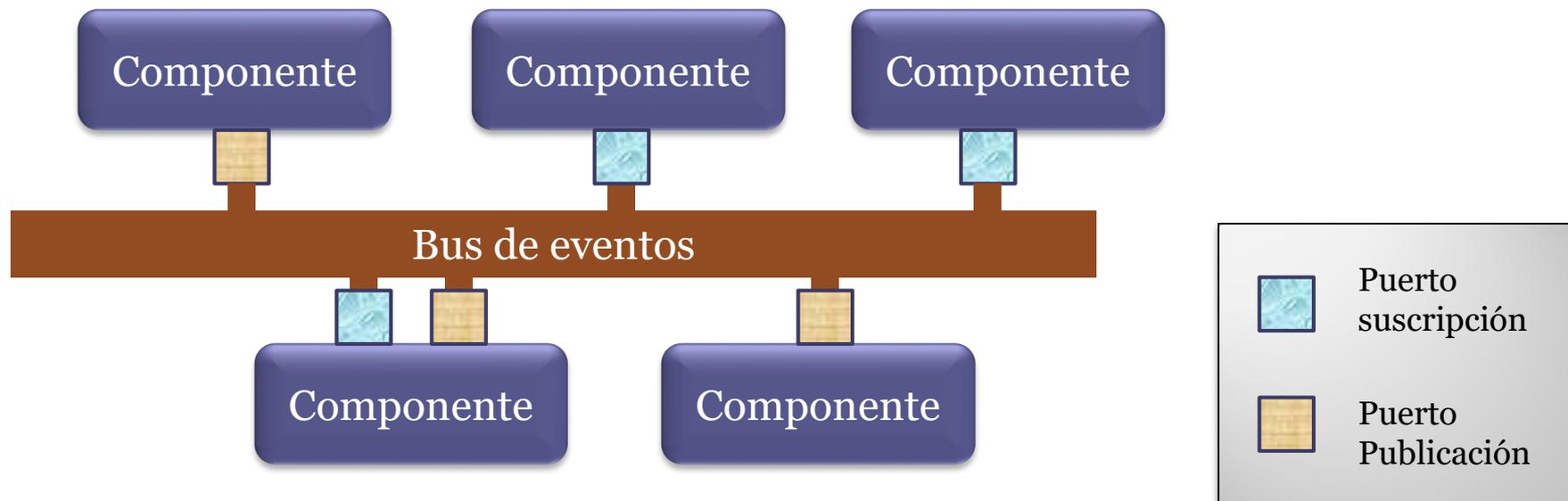
Patrones relacionados

CQRS, Event sourcing



Publish-subscribe

Componentes se subscriben a un canal para recibir mensajes de otros componentes



Publish-subscribe

Elementos:

Componente:

Componente que se suscribe al bus de eventos

Puerto de publicación

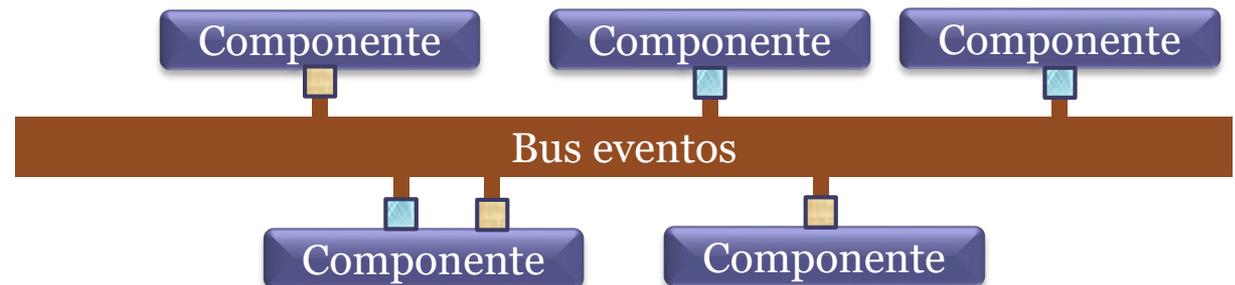
Se registra para publicar mensajes

Puerto de suscripción

Se registra para recibir cierto tipo de mensajes

Bus de eventos (canal de mensajes):

Transmite mensajes a los suscriptores



Publish-subscribe

Restricciones:

Separación puerto de suscripción/publicación

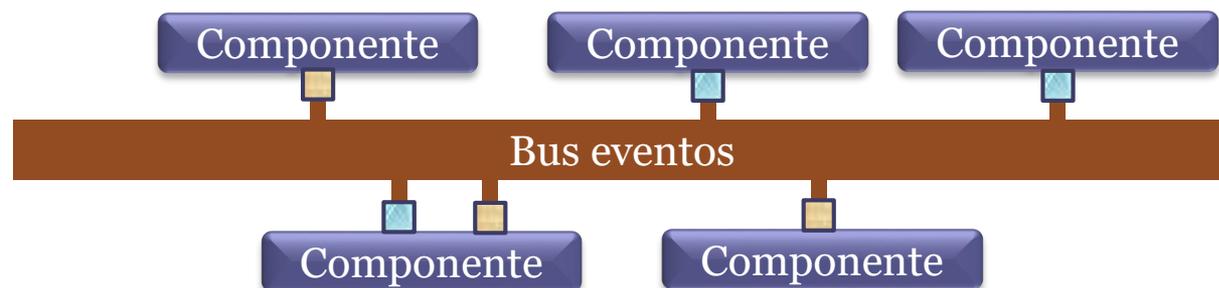
Un componente puede tener ambos puertos

Comunicación no directa

En general, comunicación asíncrona

A través de canal de mensajes

Componentes delegan responsabilidad al canal



Publish-subscribe

Ventajas

Calidad de comunicación

Mayor eficiencia

Depuración

Bajo acoplamiento

Suscriptores no dependen
de publicadores

...ni viceversa

Escalabilidad

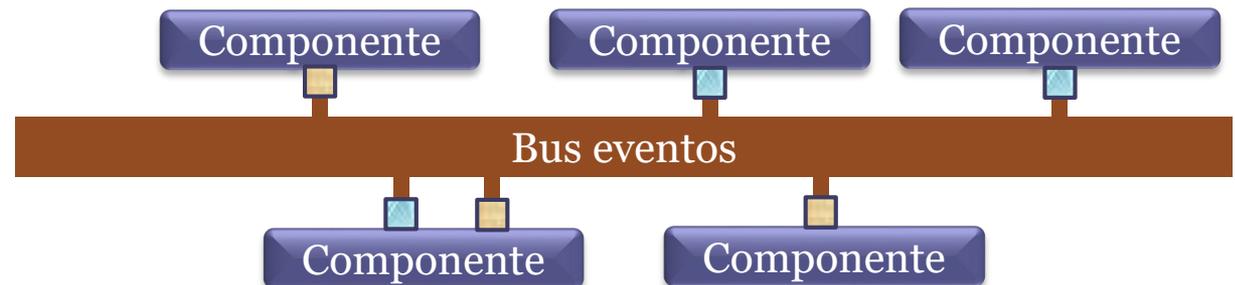
Problemas

Se añade nivel de
indirección

Comunicación directa
puede ser más
eficiente

Implementación
compleja

Puede requerir COTS



Modelos de Actores

Utilizados para computación concurrente

Actores en lugar de objetos

No hay estado compartido entre actores

Paso de mensajes asíncrono

Desarrollos teóricos desde 1973 (Carl Hewitt)

Aplicaciones en telecomunicaciones (Erlang)



Modelos de actores

Elementos

Actor: entidad computacional con estado

Se comunica con los actores enviando mensajes

Procesa los mensajes de uno en uno

Mensajes

Direcciones: Identifican a los actores (*mailing address*)



Modelos de actores

Restricciones (1)

Un actor solamente puede:

- Enviar mensajes a otros actores

 - Los mensajes son inmutables

- Crear nuevos actores

- Modificar cómo va a procesar siguiente mensaje

Los actores están desacoplados

El receptor no depende del emisor



Modelos de actores

Restricciones (2)

Paralelismo:

Todas las acciones pueden hacerse en paralelo

No hay estado global compartido

Los mensajes pueden llegar en cualquier orden

Direcciones locales

Un actor sólo puede enviar mensajes a direcciones conocidas (porque se le pasaron o porque las crea)



Modelos de actores

Ventajas

Paralelismo

Trasparencia y escalabilidad

Direcciones internas vs
externas

Modelos de actores no
locales

Servicios Web, sistemas
multi-agentes

Problemas

Envío de mensajes

Cómo garantizar que los
mensajes llegan

Coordinación entre
actores

Sistemas no
consistentes por
definición



Modelos de actores

Implementaciones

Erlang (lenguaje de programación)

Akka

Aplicaciones

Sistemas reactivos

Ejemplos: Ericsson, Facebook, twitter



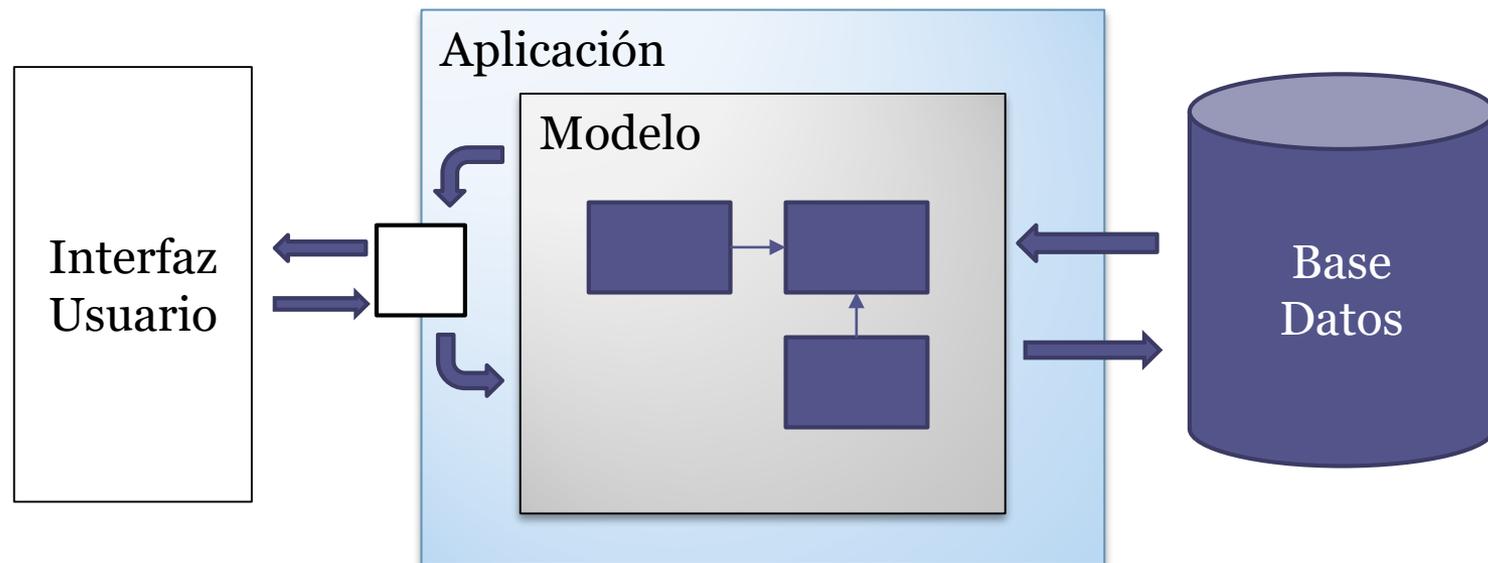
CQRS

Command Query Responsibility Seggregation

Separar el modelo en 2 partes

Command (*modificación*): Realiza cambios

Query (*consulta*): Sólo realiza consultas, actualiza interfaz



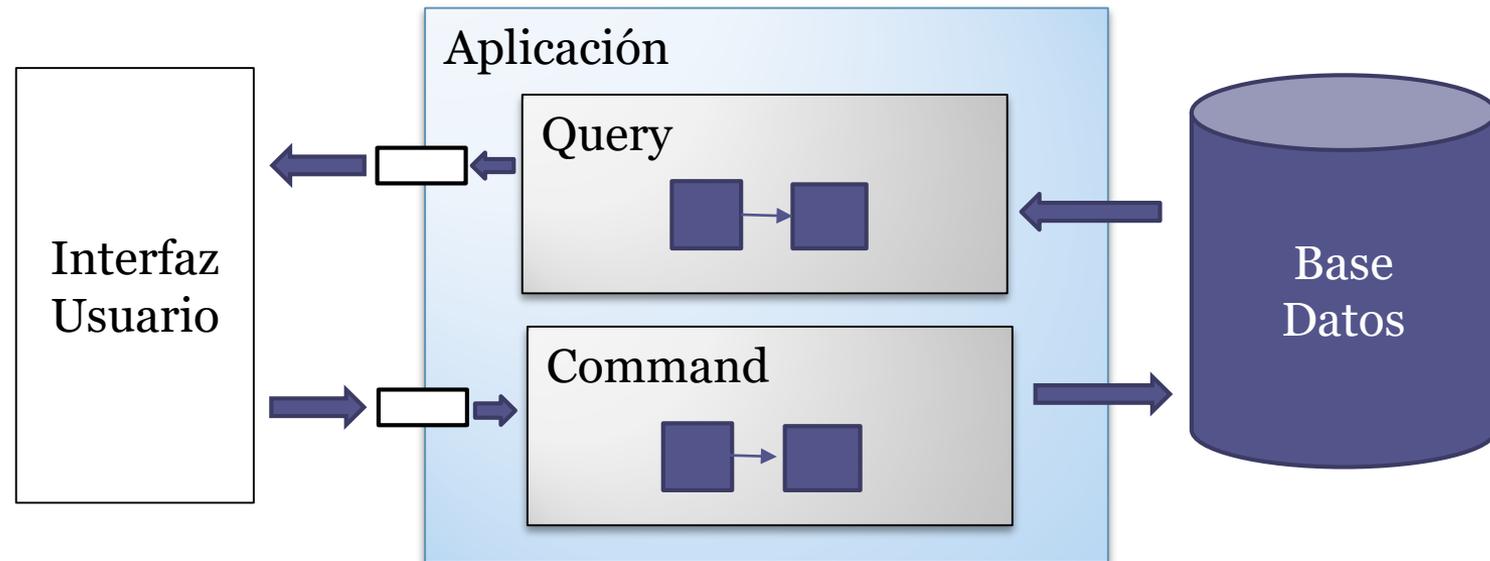
CQRS

Command Query Responsibility Seggregation

Separar el modelo en 2 partes

Command (*modificación*): Realiza cambios

Query (*consulta*): Sólo realiza consultas, actualiza interfaz



CQRS

Ventajas

Escalabilidad

Optimizar consultas
(sólo lectura)

Comandos
asíncronos

Facilita

descomposición de
equipos

Aplicaciones

Axon Framework

Problemas

Operaciones híbridas
(consulta/comando)

Ejemplo: *pop()* en una
pila

Complejidad

En entornos CRUD
puede ser excesivo

Sincronización

Posibilidad de consultas
sobre datos no
actualizados

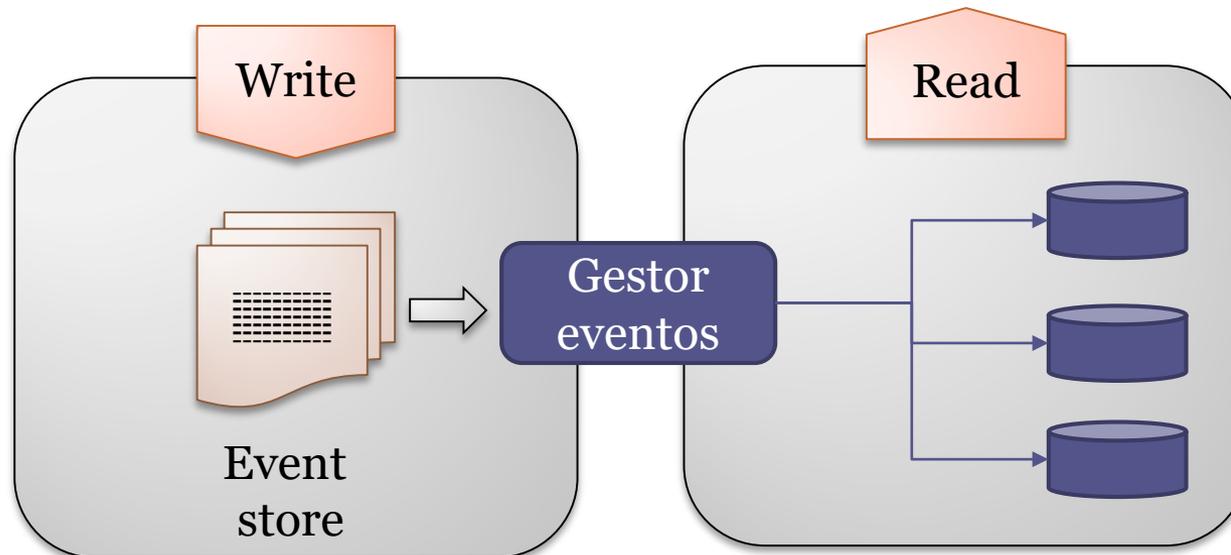
Event Sourcing

Capturar cambios del estado mediante eventos

Permite seguir traza de cómo se llegó a un determinado estado

Event Store

Siempre se añaden eventos (no se cambian)



Event Sourcing

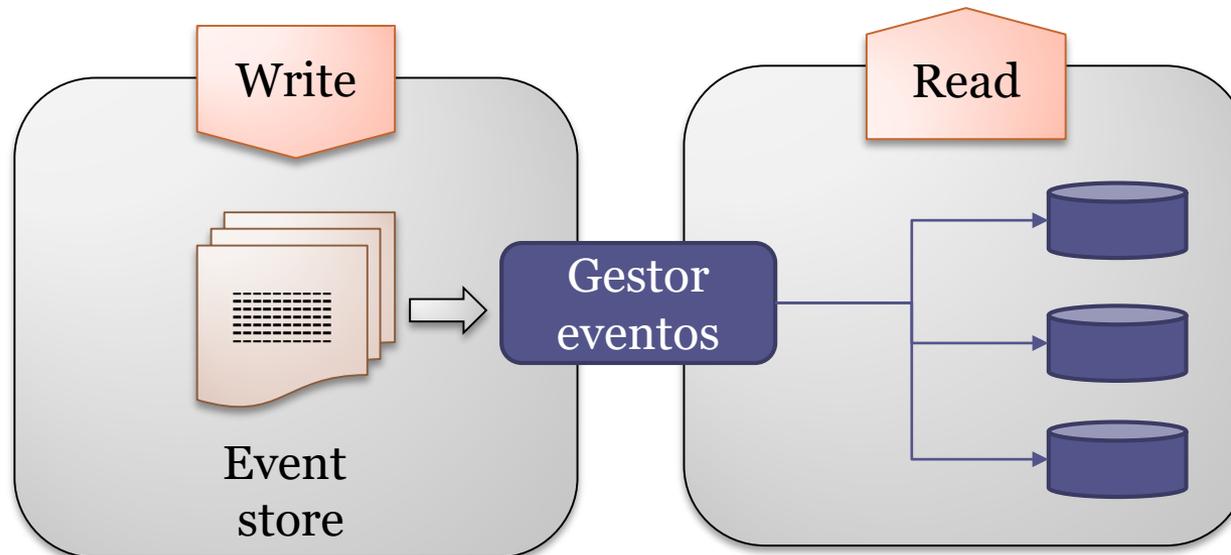
Elementos

Almacén de eventos

Almacena cambios en el estado

Eventos

No cambian, se enuncian en pasado



Event Sourcing

Ventajas

Tolerancia a fallos

Trazabilidad

Determinar estado de aplicación en cada momento

Reconstrucción

Si aparecen eventos erróneos, se pueden deshacer sus acciones y reconstruir el resto

Escalabilidad

BD de sólo append

Problemas/retos

Novedad desarrollo

Diferente de sistemas tradicionales

Consistencia de datos

Eventual consistency

Actualización software

Convivir versiones de eventos diferentes?

Gestión de recursos

Granularidad de los eventos

Almacenamiento de eventos crece con tiempo

Requiere crear instantáneas (snapshots)

Event sourcing

Aplicaciones

Sistemas de bases de datos

Datomic, EventStore

Sistemas adaptables

Plugins

Microkernel

Reflection

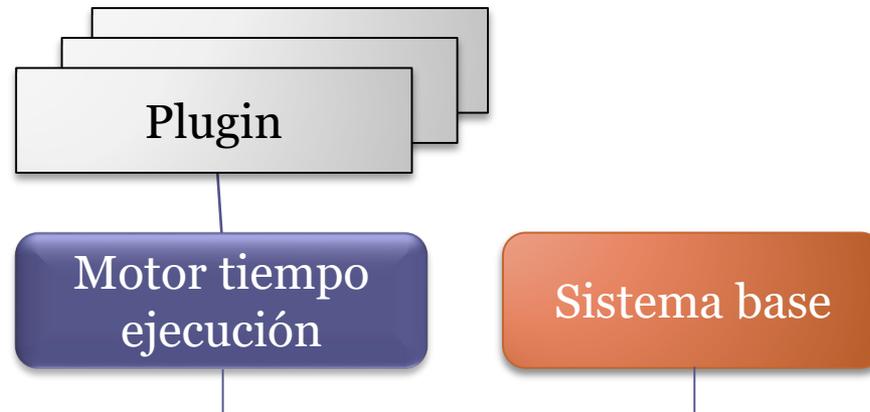
Intérpretes y DSL

Código móvil

Código bajo demanda

Plugins

Permite extender el sistema mediante la incorporación de *plugins* que añaden nuevas funcionalidades



Plugins

Elementos

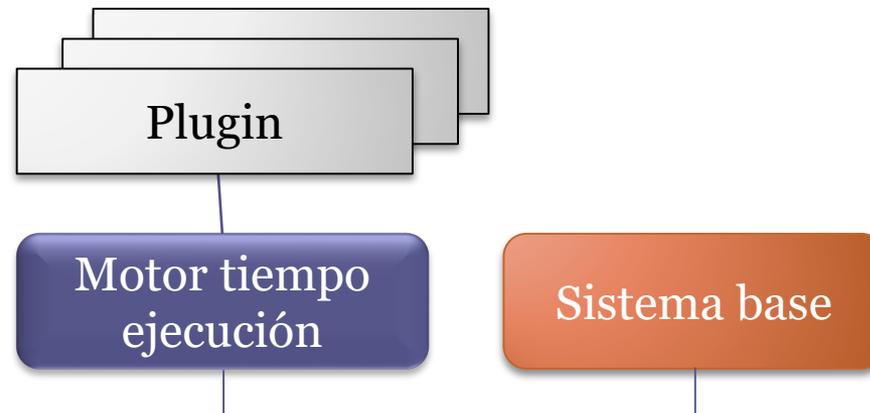
Sistema base:

Sistema que admite la incorporación de plugins

Plugins: Componentes que pueden ser añadirse o eliminarse dinámicamente

Motor de ejecución:

Arranca, localiza, inicializa y ejecuta *plugins*



Plugins

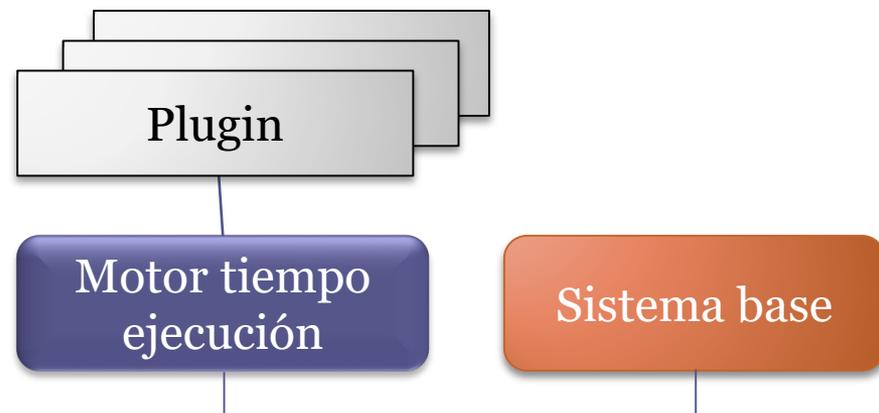
Restricciones

El motor en tiempo de ejecución se encarga de la gestión de los *plugins*

El sistema admite añadir/eliminar *plugins*

Los *plugins* pueden depender de otros

Debe declarar sus dependencias y el API que exporta



Plugins

Ventajas

Extensibilidad

Mejorar funcionamiento de aplicación de forma no prevista inicialmente

Personalización

Aplicación puede crecer bajo demanda

Adaptarse a nuevos requisitos

Problemas

Consistencia

Los plugins deben incorporarse al sistema de forma consistente

Rendimiento

Retardo en búsqueda de *plugins*

Seguridad

Plugins realizados por terceras partes

Gestión de plugins y dependencias

Plugins

Ejemplos

Eclipse

Firefox

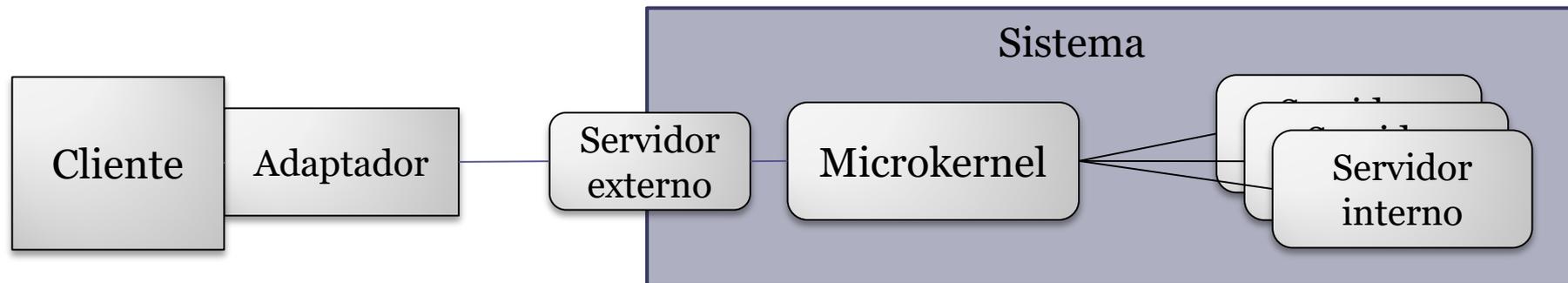
Tecnologías

Sistemas de componentes: OSGi

Microkernel

Identificar funcionalidad mínima en microkernel
La funcionalidad *extra* se implementa mediante servidores internos

La comunicación al exterior se realiza mediante servidor externo



Microkernel

Elementos

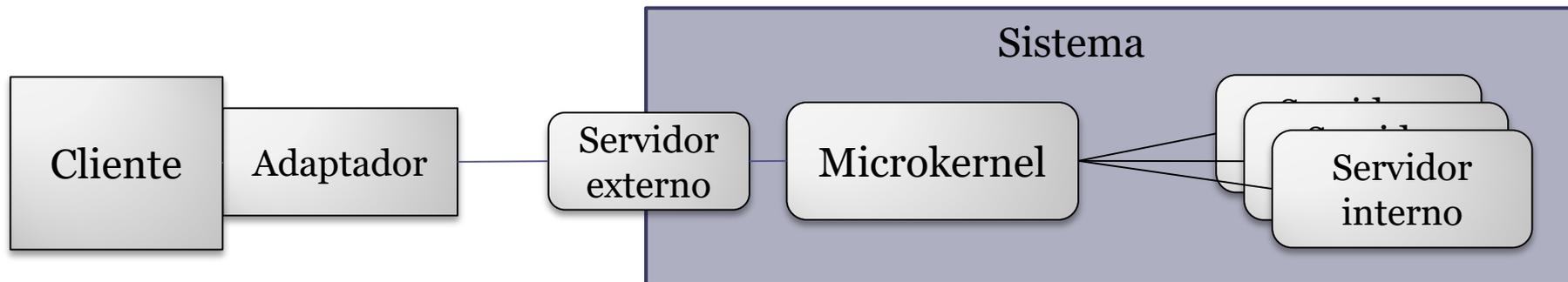
Microkernel: Funcionalidad mínima

Servidor interno: Funcionalidad extra

Servidor externo: Ofrece API externa

Cliente: Aplicación externa

Adaptador: Componente que se comunica con servidor externo



Microkernel

Restricciones:

El *microkernel* implementa solamente la funcionalidad mínima

El resto de funcionalidad es implementada por servidores internos

La comunicación del cliente con el sistema se realiza a través de los servidores externos

Microkernel

Ventajas

Portabilidad

Sólo es necesario portar el kernel

Flexibilidad y extensibilidad

Añadir nueva funcionalidad mediante nuevos servidores internos

Seguridad y fiabilidad

Encapsular partes críticas
Los errores en partes externas no afectan al microkernel

Problemas

Rendimiento

Sistema monolítico puede ser más eficiente

Complejidad de diseño

Identificar componentes del kernel
Difícil separar partes a servidores internos

Punto de fallo único

Si falla el microkernel puede comprometerse la seguridad

Microkernel

Aplicaciones

Sistemas operativos

Juegos

Editores

Reflection

Cambiar la estructura y comportamiento de una aplicación de forma dinámica

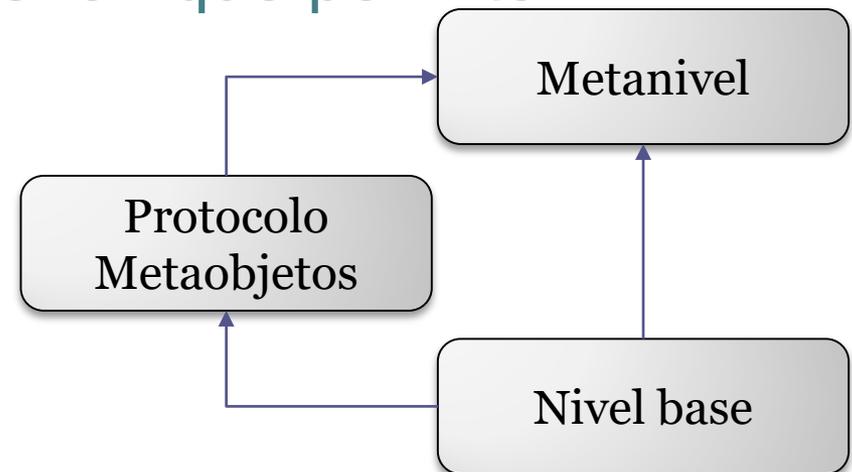
Sistemas que pueden modificarse a sí mismos

Elementos

Nivel base: Implementa lógica de la aplicación

Metanivel: Aspectos que pueden modificarse

Protocolo metaobjetos: Interfaz que permite modificar el metanivel

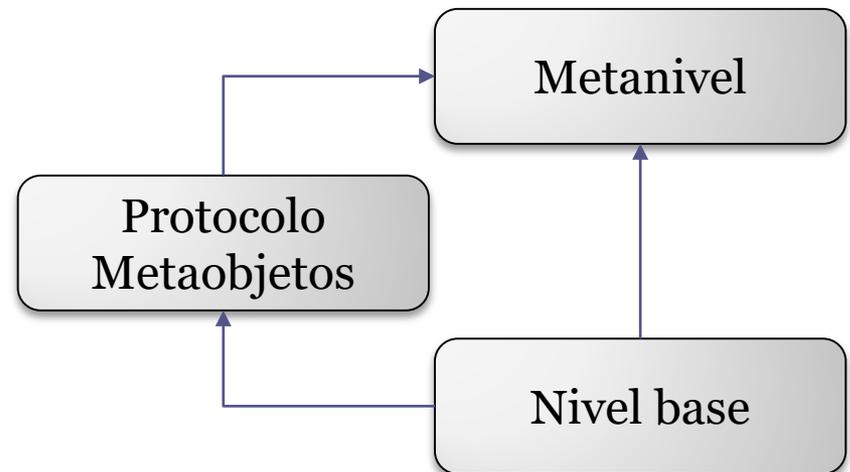


Reflection

Restricciones

El nivel base utiliza aspectos del metanivel para su funcionamiento

Durante la ejecución, el sistema puede modificar el metanivel mediante el protocolo metaobjetos



Reflection

Ventajas

Flexibilidad

El sistema puede adaptarse a condiciones cambiantes

No es necesario modificar código fuente ni detener ejecución para realizar cambios en sistema

Problemas

Implementación

No todos los lenguajes facilitan la meta-programación

Rendimiento

Puede ser necesario realizar optimizaciones para limitar la reflexividad

Seguridad:

Mantenimiento de consistencia

Reflection

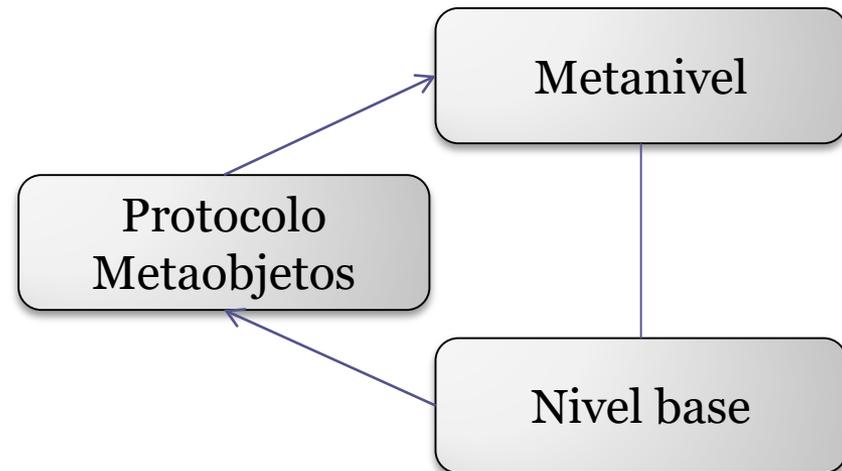
Aplicaciones

Muchos lenguajes dinámicos soportan reflectividad

Scheme, CLOS, Ruby, Python,

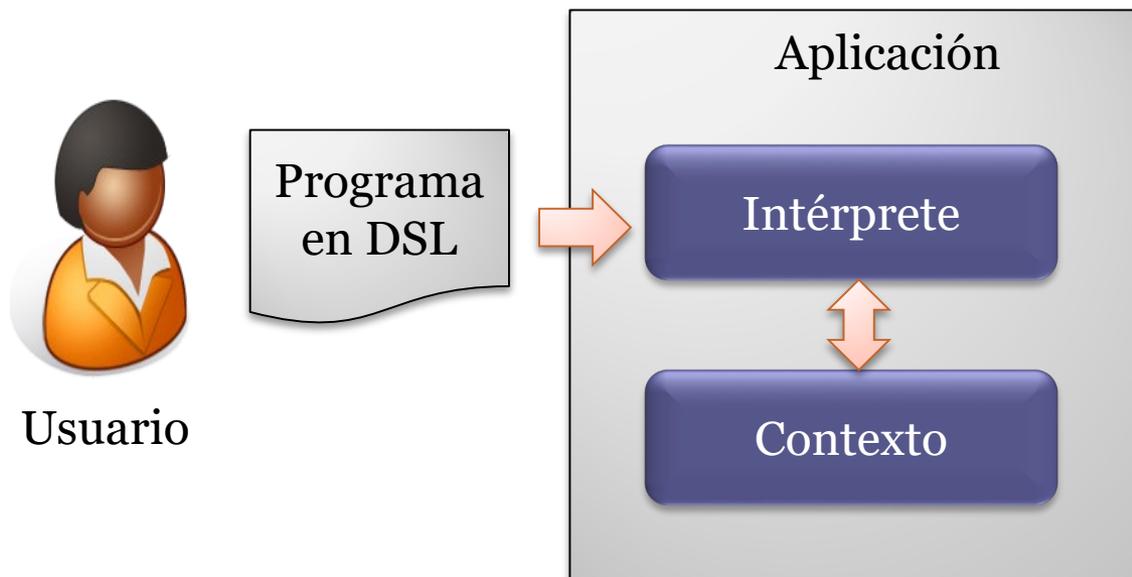
Sistemas inteligentes

Código auto-modificable



Intérpretes y DSLs

Incluyen un lenguaje de dominio específico que es interpretado por el sistema



Intérpretes y DSLs

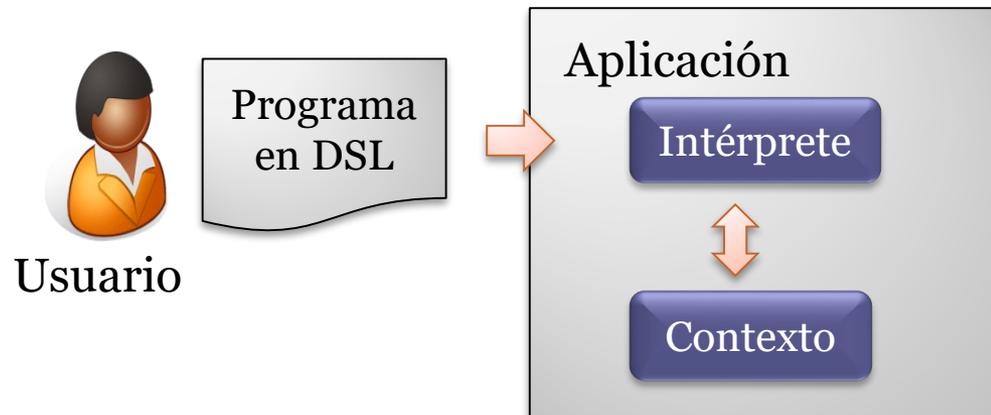
Elementos

Intérprete: Módulo que ejecuta el programa

Programa: Escrito en lenguaje de dominio específico (DSL)

El DSL puede estar pensado para que el propio usuario final pueda escribir sus programas en él

Contexto: Entorno en el que se ejecuta el programa



Intérpretes y DSLs

Restricciones

El intérprete ejecuta un programa modificando el contexto

Es necesario definir el DSL

Sintaxis (gramática)

Semántica (comportamiento)

Intérpretes y DSLs

Ventajas

Flexibilidad

Modificar

comportamiento según
necesidades del
usuario

Usabilidad

Los usuarios finales
pueden escribir sus
programas

Mayor satisfacción

Adaptabilidad

Facilidad para adaptarse
a nuevas situaciones

Problemas

Diseño del lenguaje

Sintaxis y semántica del DSL

Complejidad de implementación

Creación del intérprete
Separación de
contexto/intérprete

Rendimiento

Posibles programas no
óptimos

Seguridad

Posibles programas
incorrectos

Intérpretes y DSLs

Variaciones:

DSL empotrados

DSLs empotrados

Embedded DSLs

Lenguajes de dominio específico que están empotrados en lenguajes de alto nivel

Técnica muy utilizada en lenguajes dinámicos como Haskell, Ruby, Scala, etc.

Ventajas:

- Se reutiliza la sintaxis del lenguaje *anfitrión*

- Acceso a librerías y entornos del lenguaje anfitrión

Problemas

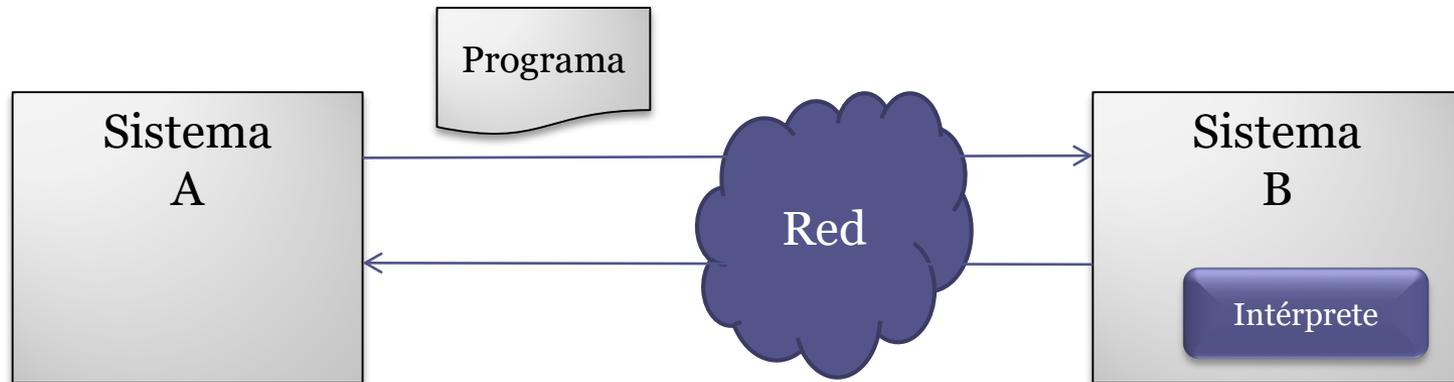
- Separación entre DSL y lenguaje anfitrión

Código móvil

Código que se transfiere de una máquina a otra

Sistema A transfiere un programa para que se ejecute en el sistema B

El sistema B debe contener un intérprete del lenguaje correspondiente



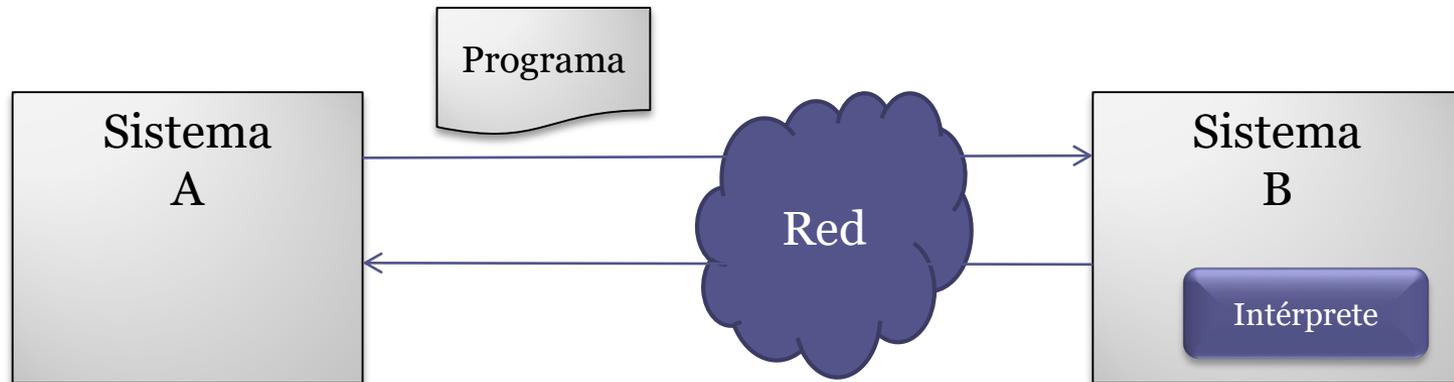
Código móvil

Elementos

Intérprete: Ejecuta el código

Programa: Código que se transfiere

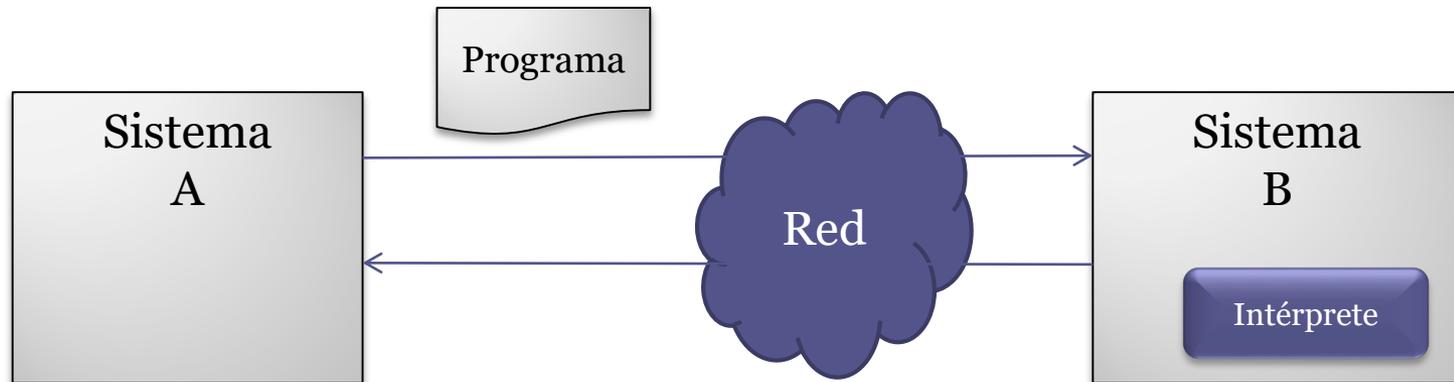
Protocolo de red: Encargado de transferir el código



Código móvil

Restricciones

- El programa debe poder ejecutarse en el sistema receptor
- El protocolo de red se encarga de transferir el programa



Código móvil

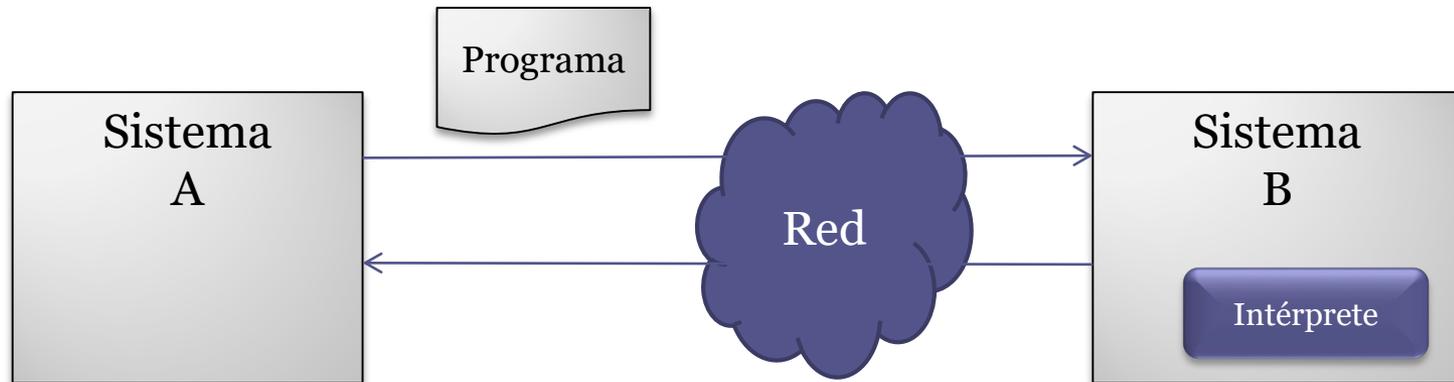
Ventajas

Flexibilidad y adaptación a diferentes entornos

Problemas

Complejidad de la implementación

Seguridad



Código móvil

Variantes

Código bajo demanda

Evaluación remota

Agentes móviles

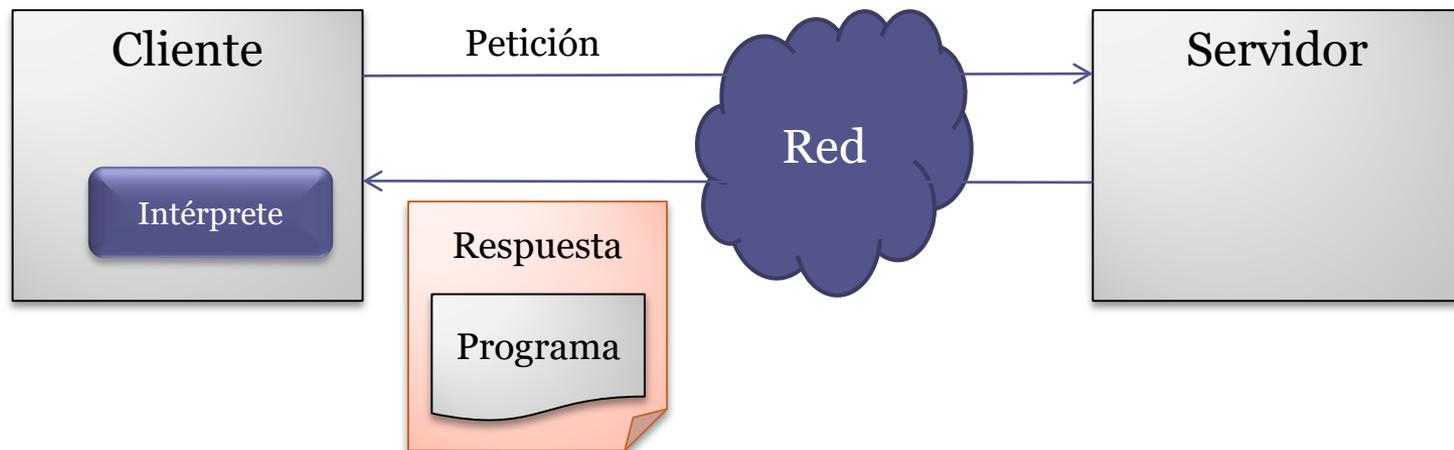
Código bajo demanda

Código es descargado y ejecutado cuando el cliente lo solicita

Combinación de código móvil y cliente-servidor

Ejemplo:

ECMAScript



Código bajo demanda

Elementos

Cliente

Servidor

Código que se transmite del servidor al cliente

Restricciones

El código reside o se genera en el servidor

Se transmite al cliente cuando el cliente lo solicita

Se ejecuta en el cliente

El cliente debe tener un intérprete del lenguaje correspondiente

Código bajo demanda

Ventajas

- Mejora experiencia de usuario
- Extensibilidad: La aplicación puede añadir nuevas funcionalidades no previstas
- No es necesario descargar o instalar la aplicación
- Concepto de *Beta permanente*
- Adaptación a entorno del cliente

Problemas

- Seguridad
- Coherencia
- Difícil garantizar comportamiento homogéneo en diferentes tipos de clientes
- El cliente puede incluso no ejecutar el programa
- Recordar: Diseño responsable

Código bajo demanda

Aplicaciones:

RIA (Rich Internet Applications)

HTML5 estandariza gran cantidad de APIs

Mejora coherencia entre clientes

Variaciones

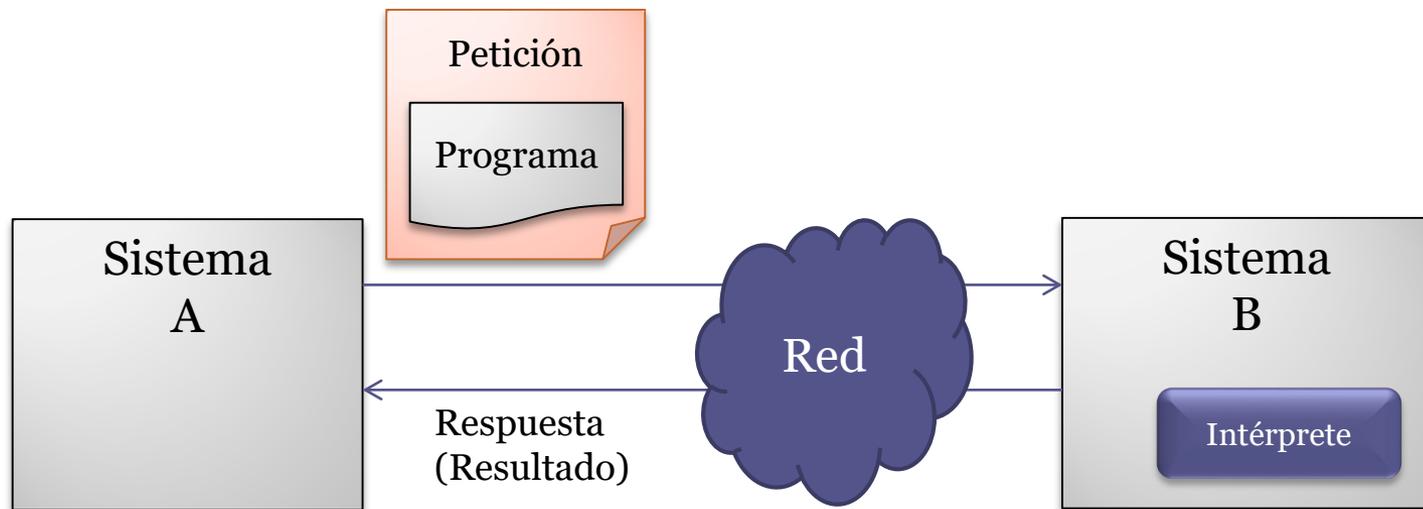
AJAX

Originalmente: *Asynchronous Javascript and XML*

El programa que se ejecuta en el cliente envía peticiones asíncronas de información al servidor sin detener su ejecución

Evaluación remota

El sistema A envía código al sistema B para que lo ejecute y le devuelva los resultados



Evaluación remota

Elementos

Sistema emisor: Realiza una petición adjuntando un programa

Sistema receptor: Ejecuta el programa y devuelve una respuesta con los resultados

Restricciones

Sistema receptor ejecuta el programa

Debe tener intérprete del lenguaje correspondiente

Protocolo de red transfiere el programa y las respuestas

Evaluación remota

Ventajas

- Aprovechar capacidades de terceras partes
- Capacidades computacionales, de memoria, etc.

Problemas

Seguridad

- Código no confiable
- Virus = variante de este estilo

Configuración

Evaluación remota

Ejemplo:

Computación voluntaria

SETI@HOME

Sirvió de base para sistema BOINC

Berkeley Open Infrastructure for Network Computing

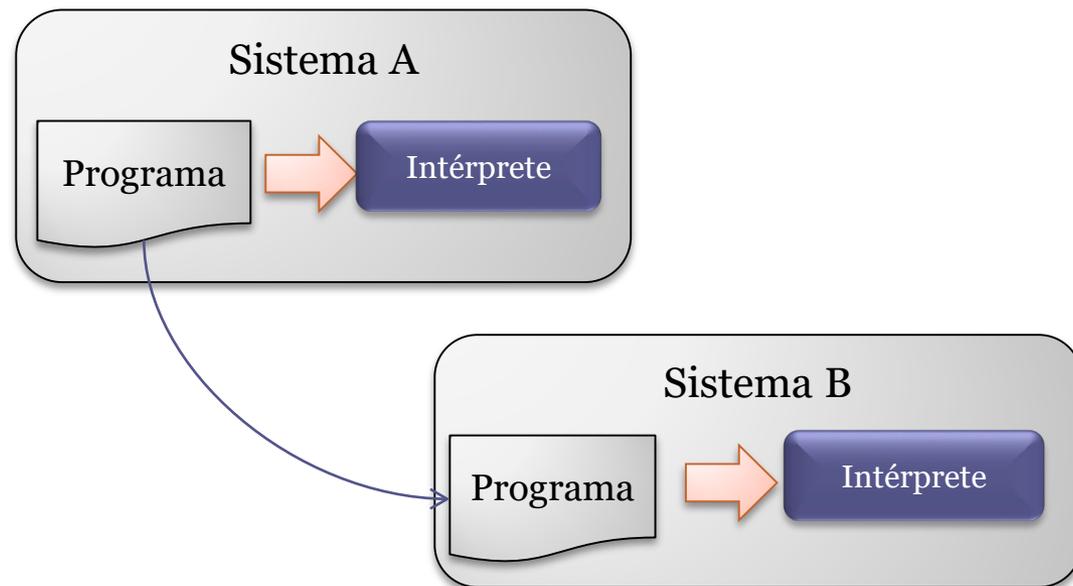
Otros proyectos: Folding@HOME, Predictor@Home,
AQUA@HOME, etc.

Agentes móviles

Código y datos pueden moverse de una máquina a otra para su ejecución

Un proceso lleva su estado de una máquina a otra

El código puede moverse de forma autónoma



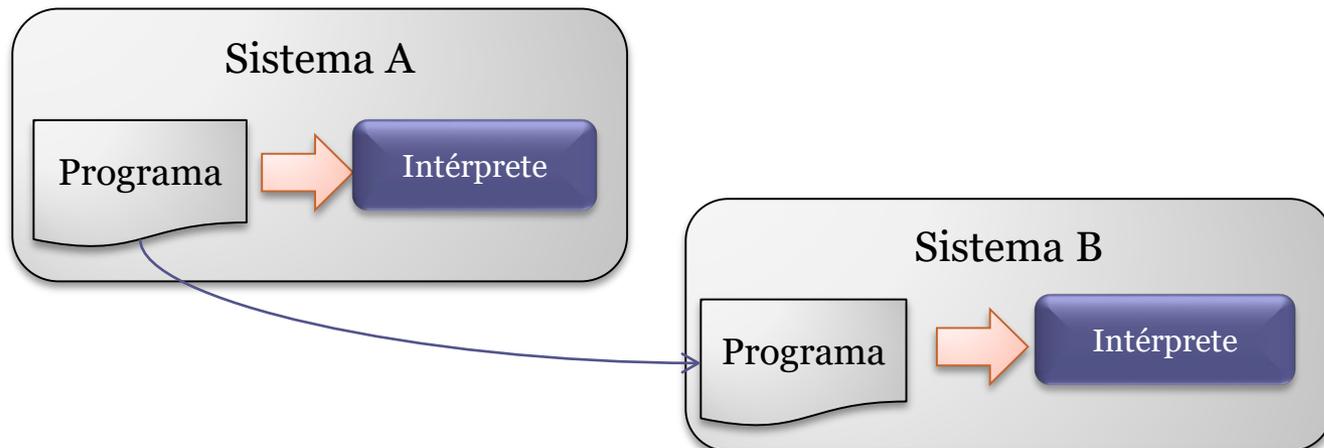
Agentes móviles

Elementos

Agente móvil: Programa que se ejecuta de un sistema a otro de forma autónoma

Sistema: Entorno de ejecución en que se ejecuta el agente móvil

Protocolo de red: transfiere el estado del agente



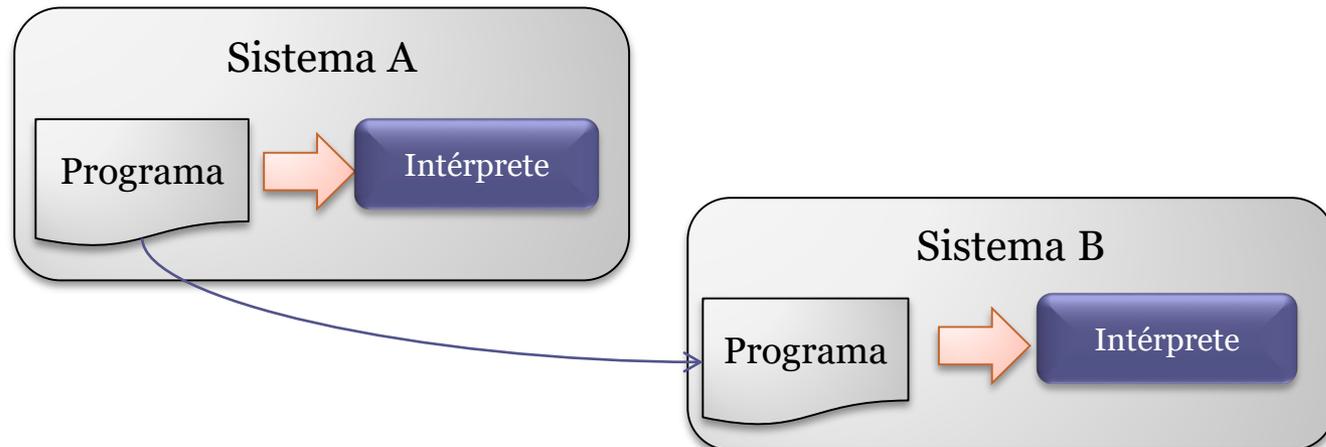
Agentes móviles

Restricciones

Los sistemas alojan y ejecutan los agentes móviles

Los agentes móviles pueden decidir cambiar su ejecución de un sistema a otro

Pueden comunicarse con otros agentes



Agentes móviles

Ventajas

Reducción de tráfico en la red

Se transmiten bloques de código que se ejecutan

Paralelismo implícito

Tolerancia a fallos de red

Conceptualmente sencillos

Agente = unidad independiente de ejecución

Posibilidad de sistemas de agentes móviles

Adaptación a cambios en el entorno

Sistemas reactivos y de aprendizaje

Problemas

Complejidad de la configuración

Seguridad

Código malicioso o erróneo

Agentes móviles

Aplicaciones

Recuperación de información

Web crawlers

Sistemas peer-to-peer

Telecomunicaciones

Control remoto y monitorización

Sistemas:

JADE (Java Agent DEvelopment framework)

Aglets de IBM

Fin