

Arquitectura de Software 75.73



**FACULTAD
DE INGENIERIA**

Universidad de Buenos Aires

Estilos Arquitectónicos #1

Christian Calónico
75.73 Arquitectura de Software

Estilos Arquitectónicos #1

1. Estilos Arquitectónicos
2. Replicated Repositories (RR)
3. Cache (\$)
4. Client-Server (CS)
5. Layered System (LS)
6. Layered Client Server (LCS)
7. Load Balancer & Reverse Proxy
8. Client-Stateless-Server (CSS)
9. Client-Cache-Stateless-Server (C\$SS)
10. Layered-Client-Cache-Stateless-Server (LC\$SS)

Estilos Arquitectónicos

Un estilo arquitectónico es un conjunto coordinado de **restricciones** arquitectónicas que restringe las funciones/características de los **elementos** arquitectónicos y las **relaciones** permitidas entre esos elementos, dentro de cualquier arquitectura que se ajusta a ese estilo



Roy Fielding

Estilos Arquitectónicos

Los estilos son un mecanismo para la **categorización** de arquitecturas y para determinar sus **características** comunes

Un estilos **encapsula decisiones importantes** sobre los elementos y hace hincapié en las **restricciones** importantes en los **elementos** y sus **relaciones**

Nuevas arquitecturas se pueden definir como **instancias de estilos** específicos

Estilos Arquitectónicos

Ya que los estilos pueden abordar **diferentes aspectos** de la arquitectura del software, una determinada arquitectura se puede **componer** de varios estilos

An architectural style [...], is applied to a design space in order to **induce** the architectural **properties** that are **desired** for the system

Algunos estilos son a menudo descritos como soluciones ***Silver Bullet*** para todos los tipos de software. Sin embargo, un buen arquitecto debe seleccionar un estilo que coincida con las **necesidades del problema particular** que se intenta resolver.

Estilos Arquitectónicos

Los estilos arquitectónicos:

- Sintetizan estructuras de soluciones arquitectónicas
- Especifican un conjunto de elementos (componentes)
- Describe sus responsabilidades
- Restringe (reglas) las relaciones entre ellos
- Permiten evaluar arquitecturas alternativas, con ventajas y desventajas conocidas ante diferentes conjuntos de atributos de calidad requeridos.

Estilos Arquitectónicos

Clasificación (Fielding):

- Data Flow Styles
- Replication Styles
- Hierarchical Styles
- Mobile Code Styles
- Peer to Peer Styles

No es una clasificación completa, sino una muestra representativa

Un nuevo estilo se puede formar sólo mediante la adición de una restricción de la arquitectura a cualquiera de los estilos presentados

Estilos Arquitectónicos

- DATA FLOW STYLES

- PIPE & FILTER
- UNIFORM PIPE & FILTER
- BATCH SEQUENTIAL

- REPLICATION STYLES

- REPLICATED REPOSITORY
- CACHE

- DATA CENTERED ARCHITECTURES

- DATABASE
- BLACKBOARD

- HIERARCHICAL STYLES

- CLIENT-SERVER
- LAYERED SYSTEMS
- LAYERED CLIENT-SERVER
- CLIENT-STATELESS-SERVER
- CLIENT-CACHE-STATELESS-SERVER
- LAYERED-CLIENT-CACHE-STATELESS-SERVER
- REMOTE SESSION
- REMOTE DATA ACCESS

- MOBILE CODE

- VIRTUAL MACHINE
- REMOTE EVALUATION
- CODE ON DEMAND
- LAYERED-CODE-ON-DEMAND-CLIENT-CACHE-STATELESS-SERVER
- MOBILE AGENT

- PEER TO PEER

- EVENT-BASED INTEGRATION
- DISTRIBUTED OBJECTS
- BROKED DISTRIBUTED OBJECTS

Estilos Arquitectónicos #1

1. Estilos Arquitectónicos
2. Replicated Repositories (RR)
3. Cache (\$)
4. Client-Server (CS)
5. Layered System (LS)
6. Layered Client Server (LCS)
7. Load Balancer & Reverse Proxy
8. Client-Stateless-Server (CSS)
9. Client-Cache-Stateless-Server (C\$SS)
10. Layered-Client-Cache-Stateless-Server (LC\$SS)

Replicated Repositories (RR)

Más de un proceso provee el mismo servicio.

Servidores descentralizados interactúan para proveer a los clientes la ilusión de que hay 1 sólo servidor centralizado.

Concepto de
duplicidad

Ej. Distributed Filesystem

Ej. CVS/SVN (local)

Replicated Repositories (RR)



Escalabilidad

Cantidad de usuarios, transacciones, manejo de picos, agregado de recursos, etc.



Performance

Reduce la latencia de requests normales

Permite trabajar en modo off-line



Confiabilidad

Una falla en un server no provoca la caída del servicio



Simplicidad

Complejidad de la replicación se compensa con la transparencia en la manipulación de datos replicados a nivel local

La principal preocupación es mantener la consistencia

Estilos Arquitectónicos #1

1. Estilos Arquitectónicos
2. Replicated Repositories (RR)
3. Cache (\$)
4. Client-Server (CS)
5. Layered System (LS)
6. Layered Client Server (LCS)
7. Load Balancer & Reverse Proxy
8. Client-Stateless-Server (CSS)
9. Client-Cache-Stateless-Server (C\$SS)
10. Layered-Client-Cache-Stateless-Server (LC\$SS)

Cache (\$)

Replicación del resultado de un request, para poder ser utilizado en posteriores requests.

Variante
del RR

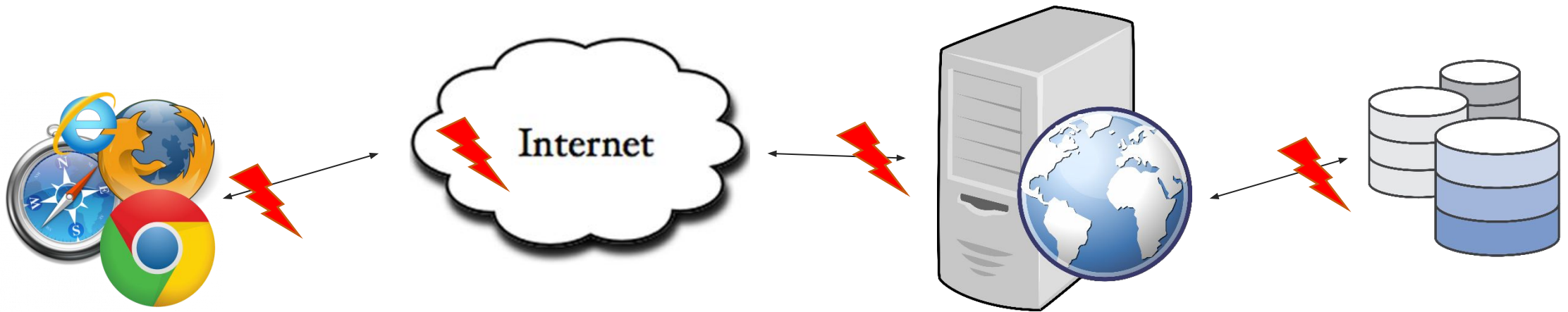
Suele ser utilizado cuando:

- El conjunto posible de datos supera las capacidades del cliente
- El acceso completo al repositorio es innecesario

- Lazy Population
- Active Population

Cache (\$)

Ej. Web Application



Cache (\$)



Performance

Mejora, pero en menor medida que en el caso de RR
Pequeña fracción de los requests impactan en cache



Simplicidad

Más fácil de implementar que RR

Con respecto a RR:

- Requiere menos recursos de procesamiento y almacenamiento
- Mayor eficiencia de red – transmite sólo cuando lo requiere
- Menores beneficios en términos de performance percibida

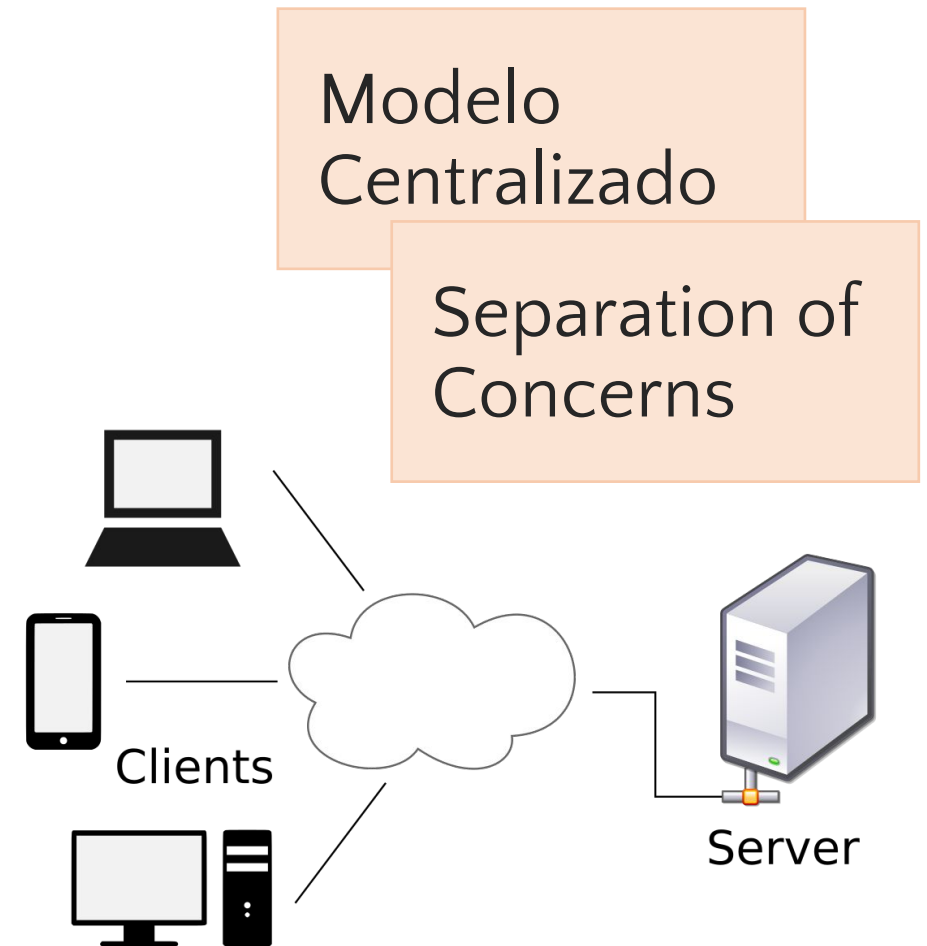
Estilos Arquitectónicos #1

1. Estilos Arquitectónicos
2. Replicated Repositories (RR)
3. Cache (\$)
4. Client-Server (CS)
5. Layered System (LS)
6. Layered Client Server (LCS)
7. Load Balancer & Reverse Proxy
8. Client-Stateless-Server (CSS)
9. Client-Cache-Stateless-Server (C\$SS)
10. Layered-Client-Cache-Stateless-Server (LC\$SS)

Client-Server (CS)

Estilo arquitectónico más frecuente para aplicaciones basadas en red

- **Servidor** (componente) – ofrece servicios, escucha peticiones
- **Cliente** (componente) – solicita un servicio (utilizando un conector)
- El servidor realiza o rechaza la solicitud, enviando una respuesta



Client-Server (CS)

- El cliente conoce al servidor; el servidor no conoce al cliente
- El servidor atiende a más de un cliente
- El cliente desencadena (y espera), el server es reactivo
- El cliente decide cuándo ocurren, el server sólo espera
- El servidor es un proceso que no termina nunca

Client-Server (CS)



Simplicidad

Causada por la separación de funciones
En general, se traduce en mover la UI al cliente



Escalabilidad

Servidor más simple, más fácil hacer crecer al sistema



Evolucionabilidad

Cliente y servidor pueden evolucionar por separado
Modificaciones en uno podrían no afectar al otro
(condicionado al mantenimiento de la interfaz entre ellos)

Estilos Arquitectónicos #1

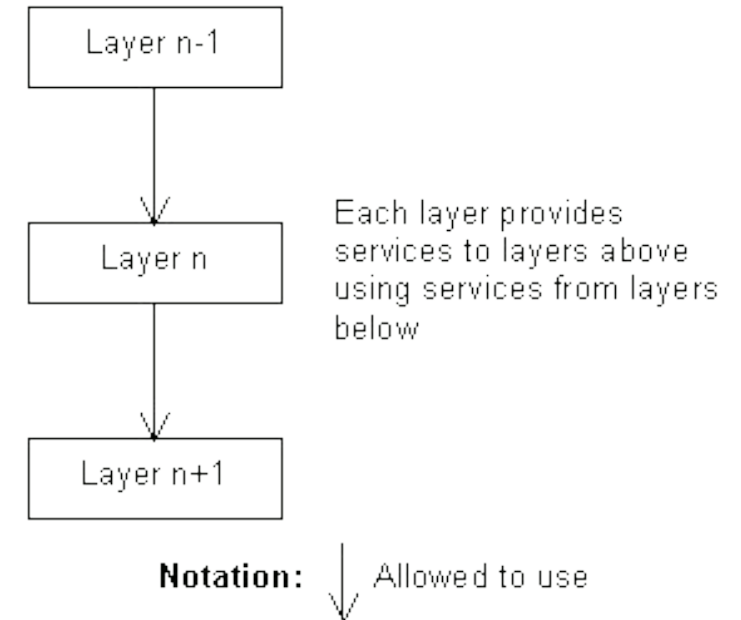
1. Estilos Arquitectónicos
2. Replicated Repositories (RR)
3. Cache (\$)
4. Client-Server (CS)
5. Layered System (LS)
6. Layered Client Server (LCS)
7. Load Balancer & Reverse Proxy
8. Client-Stateless-Server (CSS)
9. Client-Cache-Stateless-Server (C\$SS)
10. Layered-Client-Cache-Stateless-Server (LC\$SS)

Layered System (LS)






Sistema de capas, organizadas jerárquicamente

Cada capa brinda servicios a su capa superior
Cada capa usa servicios de su capa inferior

Reducen el acoplamiento, ocultando las capas internas



Layered System (LS)

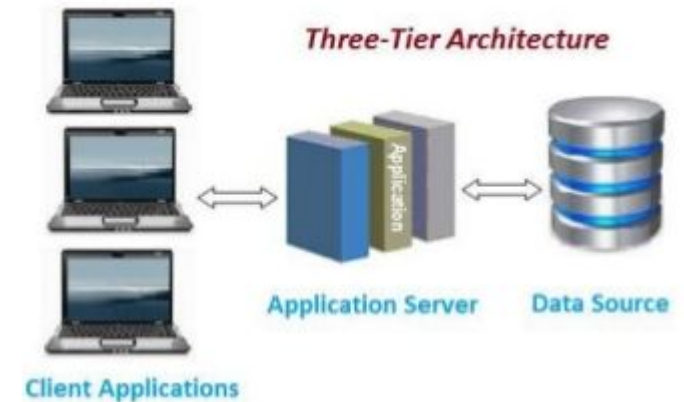
-  Evolucionabilidad Se puede evolucionar un layer, sin afectar a otros
-  Extensibilidad Se puede agregar un nuevo layer
-  Performance Adición de overhead y latencia
-  Reusabilidad Se pueden reutilizar layers
-  Portabilidad Se pueden portar layers (ej. TCP/IP)

Estilos Arquitectónicos #1

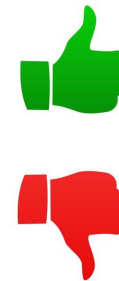
1. Estilos Arquitectónicos
2. Replicated Repositories (RR)
3. Cache (\$)
4. Client-Server (CS)
5. Layered System (LS)
6. Layered Client Server (LCS)
7. Load Balancer & Reverse Proxy
8. Client-Stateless-Server (CSS)
9. Client-Cache-Stateless-Server (C\$SS)
10. Layered-Client-Cache-Stateless-Server (LC\$SS)

Layered Client Server (LCS)

En sistemas basados en red, la aplicación de Layers se limita a su combinación con CS, a fin de proporcionar *CS en capas*



Las arquitecturas CS en capas se conocen también como arquitecturas de dos niveles, tres niveles, o de multi-niveles (N-Tier)



Suma de
CS y LS

Layered Client Server (LCS)

Reverse Proxy – layer de aplicación, accesible desde varios clientes, que recibe *requests* y los reenvía (con posibles traducciones) a componentes *servidor*

Estilos Arquitectónicos #1

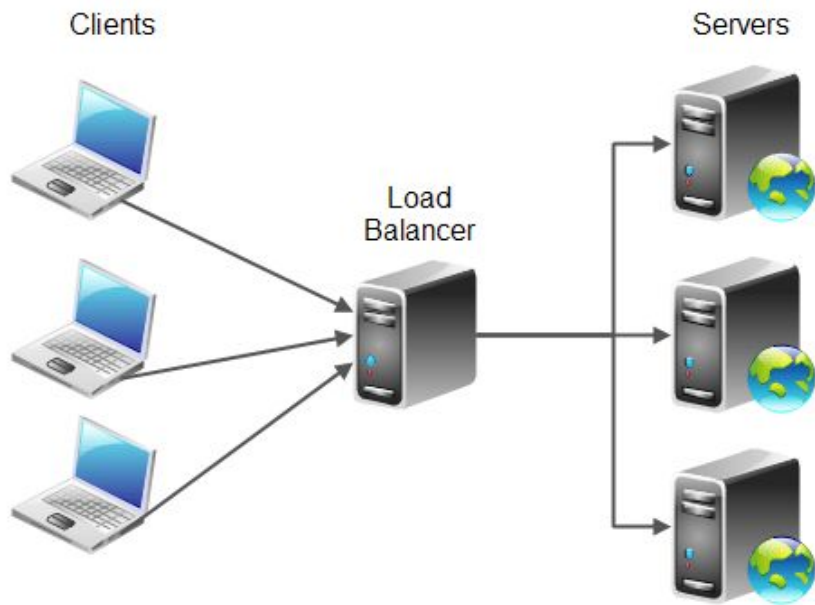
1. Estilos Arquitectónicos
2. Replicated Repositories (RR)
3. Cache (\$)
4. Client-Server (CS)
5. Layered System (LS)
6. Layered Client Server (LCS)
7. Load Balancer & Reverse Proxy
8. Client-Stateless-Server (CSS)
9. Client-Cache-Stateless-Server (C\$SS)
10. Layered-Client-Cache-Stateless-Server (LC\$SS)

Load Balancer & Reverse Proxy

- Ambos son *componentes* en una arquitectura cliente-servidor.
- Ambos actúan como *intermediarios* entre clientes y servidores.
- Ambos realizan funciones que mejoran la *eficiencia*.

- Pueden utilizarse *dispositivos específicos, dedicados*.
- Sin embargo, en las arquitecturas modernas se utilizan aplicaciones de *software*, que se ejecutan en *commodity hardware*.

Load Balancer



Load Balancer

Recibe requests de los clientes, y los distribuye entre un grupo de servidores, direccionando luego la respuesta del server seleccionado al cliente apropiado

Usualmente los servidores brindan el mismo servicio, aunque se podrían tratar ciertos requests de forma diferenciada

Load Balancer

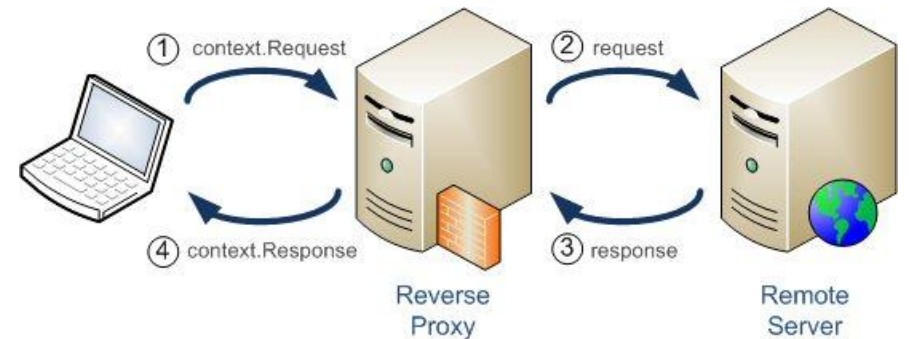
- Soporta un volumen de usuarios/requests más alto (scalability)
- Según el caso, podría escalar acorde a la demanda (elasticity)
- Evita una caída por sobrecarga (availability)
- Monitorea y gestiona los servidores disponibles (availability)
- Elimina un único punto de falla (reliability)
- Mejor uso de los recursos escasos (performance)
- Soluciones específicas según tipo de request (performance)
- Reduce el número de errores visibles por el usuario (UX)

Reverse Proxy

Reverse Proxy

Recibe requests de los clientes, los reenvía a un servidor que pueda realizarlo, y le retorna la respuesta del servidor al cliente (esconde la identidad del servidor)

- Actúa como *Public Face*
- ¿Tiene sentido con 1 único servidor?



Reverse Proxy

- Protección ante ataques informáticos (security)
 - Esconde a los servidores (no se accede, evita explotar vulnerabilidades)
 - Protección contra ataques DoS (ej. blacklist, límite de conexiones, etc.)
- Web acceleration (performance)
 - Ej. caching, compression, SSL termination
- Favorece la elasticidad (scalability)
 - El cliente sólo conoce la IP del proxy
 - La configuración del backend puede cambiar acorde a la demanda

Load Balancer & Reverse Proxy

Cluster (de servidores)

Unión de varios servidores que trabajan como si de uno solo se tratasen

- ¿Tiene sentido usar varias instancias de un servicio en un mismo server físico?
- ¿Cómo impactan los locks en la BD?

Estilos Arquitectónicos #1

1. Estilos Arquitectónicos
2. Replicated Repositories (RR)
3. Cache (\$)
4. Client-Server (CS)
5. Layered System (LS)
6. Layered Client Server (LCS)
7. Load Balancer & Reverse Proxy
8. Client-Stateless-Server (CSS)
9. Client-Cache-Stateless-Server (C\$SS)
10. Layered-Client-Cache-Stateless-Server (LC\$SS)

Client Stateless Server (CSS)

Deriva de CS, pero con una restricción adicional: **no** se permite almacenar el ***estado de la sesión*** en el componente servidor

Los *requests* del cliente deben contener **toda** la información necesaria para ser comprendidos

- El servidor no mantiene ningún tipo de contexto
- El estado se mantiene totalmente en el cliente

¿Qué es el estado?

Idempotencia

Client Stateless Server (CSS)



Visibilidad

Simplifica la tarea de un sistema de monitoreo



Performance

Transmisión de datos repetitivos a través de la red



Confiabilidad

Facilita la recuperación ante fallos parciales



Escalabilidad

Permite asignar libremente un servidor

¿Cómo impacta CSS en un Load Balancer?



Estilos Arquitectónicos #1

1. Estilos Arquitectónicos
2. Replicated Repositories (RR)
3. Cache (\$)
4. Client-Server (CS)
5. Layered System (LS)
6. Layered Client Server (LCS)
7. Load Balancer & Reverse Proxy
8. Client-Stateless-Server (CSS)
9. Client-Cache-Stateless-Server (C\$SS)
10. Layered-Client-Cache-Stateless-Server (LC\$SS)

Client Cache Stateless Server (C\$SS)

Deriva de CSS y de \$.
Agrega mediadores de cache.

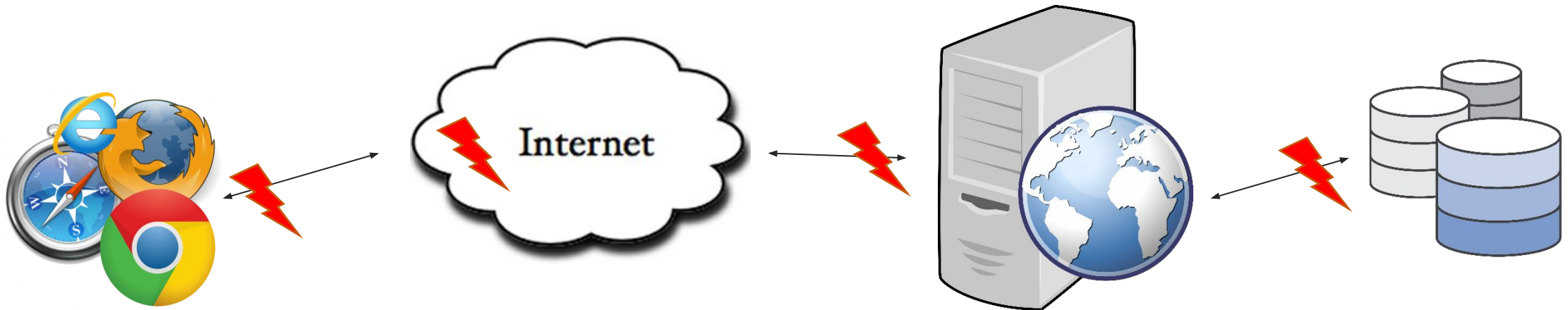
¿Dónde?



Performance



Eficiencia



Estilos Arquitectónicos #1

1. Estilos Arquitectónicos
2. Replicated Repositories (RR)
3. Cache (\$)
4. Client-Server (CS)
5. Layered System (LS)
6. Layered Client Server (LCS)
7. Load Balancer & Reverse Proxy
8. Client-Stateless-Server (CSS)
9. Client-Cache-Stateless-Server (C\$SS)
10. Layered-Client-Cache-Stateless-Server (LC\$SS)

Layered Client Cache Stateless Server (LC\$SS)

Deriva de LCS y de C\$SS.
Agrega componentes proxy y cache.



Suma de LCS
y C\$SS

Estilos Arquitectónicos #1

1. Estilos Arquitectónicos
2. Replicated Repositories (RR)
3. Cache (\$)
4. Client-Server (CS)
5. Layered System (LS)
6. Layered Client Server (LCS)
7. Load Balancer & Reverse Proxy
8. Client-Stateless-Server (CSS)
9. Client-Cache-Stateless-Server (C\$SS)
10. Layered-Client-Cache-Stateless-Server (LC\$SS)

Consultas?

Feedback

<https://goo.gl/forms/NvrORS12kuuBitpE3>

Christian Calónico
ccalónico@fi.uba.ar

Estilos Arquitectónicos #2

Christian Calónico
75.73 Arquitectura de Software

Estilos Arquitectónicos #2

1. **Recapitulando**
2. Remote Session (RS)
3. Remote Data Access (RDA)
4. Mobile Code Styles
5. Virtual Machine (VM)
6. Remote Evaluation (REV)
7. Code on Demand (COD)
8. Layered Code on Demand Client
Cache Stateless Server (LCODC\$SS)
9. Mobile Agent (MA)

Estilos Arquitectónicos

- DATA FLOW STYLES

- PIPE & FILTER
- UNIFORM PIPE & FILTER
- BATCH SEQUENTIAL

- REPLICATION STYLES

- REPLICATED REPOSITORY
- CACHE

- DATA CENTERED ARCHITECTURES

- DATABASE
- BLACKBOARD

- HIERARCHICAL STYLES

- CLIENT-SERVER
- LAYERED SYSTEMS
- LAYERED CLIENT-SERVER
- CLIENT-STATELESS-SERVER
- CLIENT-CACHE-STATELESS-SERVER
- LAYERED-CLIENT-CACHE-STATELESS-SERVER
- REMOTE SESSION
- REMOTE DATA ACCESS

- MOBILE CODE

- VIRTUAL MACHINE
- REMOTE EVALUATION
- CODE ON DEMAND
- LAYERED-CODE-ON-DEMAND-CLIENT-CACHE-STATELESS-SERVER
- MOBILE AGENT

- PEER TO PEER

- EVENT-BASED INTEGRATION
- DISTRIBUTED OBJECTS
- BROKERED DISTRIBUTED OBJECTS

Estilos Arquitectónicos #2

1. Recapitulando
2. Remote Session (RS)
3. Remote Data Access (RDA)
4. Mobile Code Styles
5. Virtual Machine (VM)
6. Remote Evaluation (REV)
7. Code on Demand (COD)
8. Layered Code on Demand Client
Cache Stateless Server (LCODC\$SS)
9. Mobile Agent (MA)

Remote Session (RS)

Deriva de CS, intentando minimizar la complejidad y maximizar la reutilización de los componentes del cliente

- Los clientes inician sesión en el servidor, invocan sus servicios, y luego finalizan la sesión
- El estado de la aplicación es mantenido totalmente en el servidor

Ejemplos:

- Telnet (cliente genérico)
- FTP

Remote Session (RS)



Simplicidad

Es más fácil mantener centralizadamente la interfaz del servidor



Escalabilidad

El server requiere mantener el estado de la app



Visibilidad

El monitor debería conocer el estado completo del servidor

Estilos Arquitectónicos #2

1. Recapitulando
2. Remote Session (RS)
3. Remote Data Access (RDA)
4. Mobile Code Styles
5. Virtual Machine (VM)
6. Remote Evaluation (REV)
7. Code on Demand (COD)
8. Layered Code on Demand Client
Cache Stateless Server (LCODC\$SS)
9. Mobile Agent (MA)

Remote Data Access (RDA)

Deriva de CS, distribuyendo el estado de la aplicación entre el cliente y el servidor

- El cliente envía una consulta a un servidor remoto, en un formato estándar (ej. SQL)
- El servidor crea un **workspace** (espacio de trabajo) y ejecuta la consulta (pudiendo dar lugar a un conjunto muy grande de datos)
- Luego, el cliente puede realizar más operaciones sobre el dataset (ej. unión de tablas, o recorrer entradas)

Remote Data Access (RDA)



Eficiencia

Datasets grandes pueden ser reducidos del lado del servidor, sin transmisión de datos



Escalabilidad

El server mantiene parte del estado de la app



Confiabilidad

Si una operación falla, el workspace puede quedar en un estado desconocido



Visibilidad

Utilización de un lenguaje estándar (ej. SQL)

... pero el monitor necesitaría conocer el estado...

Estilos Arquitectónicos #2

1. Recapitulando
2. Remote Session (RS)
3. Remote Data Access (RDA)
4. Mobile Code Styles
5. Virtual Machine (VM)
6. Remote Evaluation (REV)
7. Code on Demand (COD)
8. Layered Code on Demand Client
Cache Stateless Server (LCODC\$SS)
9. Mobile Agent (MA)

Mobile Code Styles

- Se utiliza el concepto de “movilidad” para cambiar dinámicamente la distancia entre el procesamiento y las fuentes/destinos de los datos (entradas/resultados)
- Se presta atención a la ubicación de los componentes, dado que impactan en el costo de las interacciones entre ellos
 - + Eficiencia
 - + Performance percibida
- En estos estilos, un elemento de datos se transforma dinámicamente en un componente

Estilos Arquitectónicos #2

1. Recapitulando
2. Remote Session (RS)
3. Remote Data Access (RDA)
4. Mobile Code Styles
5. Virtual Machine (VM)
6. Remote Evaluation (REV)
7. Code on Demand (COD)
8. Layered Code on Demand Client
Cache Stateless Server (LCODC\$SS)
9. Mobile Agent (MA)

Virtual Machine (VM)

La máquina virtual (o intérprete) es la base de todos los estilos de Mobile Code

- Se ejecuta código en un ambiente controlado
- Busca satisfacer requerimientos de seguridad y confiabilidad

Ej. Motor de lenguaje de scripting (ej. Perl)

Virtual Machine (VM)



Portabilidad

Separa el código de su implementación/aplicación en una plataforma específica



Visibilidad

Es difícil saber que va a hacer un ejecutable mirando sólo su código fuente



Extensibilidad

Es más fácil crear funcionalidad sobre una plataforma genérica

Estilos Arquitectónicos #2

1. Recapitulando
2. Remote Session (RS)
3. Remote Data Access (RDA)
4. Mobile Code Styles
5. Virtual Machine (VM)
6. Remote Evaluation (REV)
7. Code on Demand (COD)
8. Layered Code on Demand Client
Cache Stateless Server (LCODC\$SS)
9. Mobile Agent (MA)

Remote Evaluation (REV)

Deriva de VM + CS

Ejemplo?

SQL

- El cliente tiene el know-how para ejecutar un servicio
- El cliente no tiene los recursos (CPU, memoria, data source, etc.)
- El cliente envía “el conocimiento” al servidor, quien lo ejecuta (utilizando sus recursos) y devuelve el resultado al cliente
- Entorno protegido, sin afectar a otros clientes

Remote Evaluation (REV)



Visibilidad

El cliente envía código en lugar de un simple request



Escalabilidad

A mayor carga, mayor consumo de recursos



Extensibilidad

Se customizan los componentes del servidor



Eficiencia

Hacer la acción del lado del cliente sería más costoso



Confiabilidad

El servidor tiene menos control de la ejecución

Estilos Arquitectónicos #2

1. Recapitulando
2. Remote Session (RS)
3. Remote Data Access (RDA)
4. Mobile Code Styles
5. Virtual Machine (VM)
6. Remote Evaluation (REV)
7. Code on Demand (COD)
8. Layered Code on Demand Client
Cache Stateless Server (LCODC\$SS)
9. Mobile Agent (MA)

Code on Demand (COD)

Deriva de VM + CS

Ejemplo?

JS

- El cliente tiene los recursos para ejecutar el servicio (CPU, memoria, data source, etc.)
- El cliente no tiene el know-how
- El cliente envía un request al servidor, pidiéndole el código que representa el “know-how”, lo recibe, y lo ejecuta localmente

Code on Demand (COD)



Configurabilidad

Permite agregar features a un cliente ya “deployado”



Extensibilidad

Ídem anterior



Escalabilidad

Se utilizan los recursos de los clientes, no del server



Performance

La ejecución se realiza interactuando localmente con el cliente, sin necesidad del servidor



Visibilidad

El servidor envía código en lugar de datos

Estilos Arquitectónicos #2

1. Recapitulando
2. Remote Session (RS)
3. Remote Data Access (RDA)
4. Mobile Code Styles
5. Virtual Machine (VM)
6. Remote Evaluation (REV)
7. Code on Demand (COD)
8. Layered Code on Demand Client
Cache Stateless Server (LCODC\$SS)
9. Mobile Agent (MA)

Layered Code on Demand Client Cache Stateless Server (LCODC\$SS)

Deriva de LC\$SS + COD

- Más cerca del concepto de Web actual
- Ej. Browsers que permiten la ejecución de applets y extensiones al protocolo
- Notar que este estilo no implica HTTP, REST, ni nada relacionado

Estilos Arquitectónicos #2

1. Recapitulando
2. Remote Session (RS)
3. Remote Data Access (RDA)
4. Mobile Code Styles
5. Virtual Machine (VM)
6. Remote Evaluation (REV)
7. Code on Demand (COD)
8. Layered Code on Demand Client
Cache Stateless Server (LCODC\$SS)
9. Mobile Agent (MA)

Mobile Agent (MA)

Deriva de REV + COD

Ejemplo?

????

- Un componente entero es movido de un sitio remoto, incluyendo su estado, código y datos
- Funciona en ambos sentidos (c→s, s→c)
- Gran flexibilidad respecto a cuándo hacer el movimiento
- Una aplicación podría estar procesando algo y decidir moverse a otra ubicación, quizá para estar más cerca del próximo dataset

Estilos Arquitectónicos #2

1. Recapitulando
2. Remote Session (RS)
3. Remote Data Access (RDA)
4. Mobile Code Styles
5. Virtual Machine (VM)
6. Remote Evaluation (REV)
7. Code on Demand (COD)
8. Layered Code on Demand Client
Cache Stateless Server (LCODC\$SS)
9. Mobile Agent (MA)

Consultas?

Feedback

<https://goo.gl/forms/NvrORS12kuuBitpE3>

Christian Calónico
ccalónico@fi.uba.ar

Estilos Arquitectónicos #3

Christian Calónico
75.73 Arquitectura de Software

Estilos Arquitectónicos

- DATA FLOW STYLES

- PIPE & FILTER
- UNIFORM PIPE & FILTER
- BATCH SEQUENTIAL

- REPLICATION STYLES

- REPLICATED REPOSITORY
- CACHE

- DATA CENTERED ARCHITECTURES

- DATABASE
- BLACKBOARD

- HIERARCHICAL STYLES

- CLIENT-SERVER
- LAYERED SYSTEMS
- LAYERED CLIENT-SERVER
- CLIENT-STATELESS-SERVER
- CLIENT-CACHE-STATELESS-SERVER
- LAYERED-CLIENT-CACHE-STATELESS-SERVER
- REMOTE SESSION
- REMOTE DATA ACCESS

- MOBILE CODE

- VIRTUAL MACHINE
- REMOTE EVALUATION
- CODE ON DEMAND
- LAYERED-CODE-ON-DEMAND-CLIENT-CACHE-STATELESS-SERVER
- MOBILE AGENT

- PEER TO PEER STYLES

- EVENT-BASED INTEGRATION
- DISTRIBUTED OBJECTS
- BROKERED DISTRIBUTED OBJECTS

Estilos Arquitectónicos #3

PEER TO PEER STYLES

1. Event Based Integration (EBI)
2. Distributed Objects (DO)
3. Brokered Distributed Objects (BDO)

DATA CENTERED ARCHITECTURES

4. Database
5. Blackboard

DATA FLOW STYLES

6. Pipe & Filter (PF)
7. Uniform Pipe & Filter (UPF)
8. Batch Sequential

Event Based Integration (EBI)

- Estilo *Peer to Peer*
- También conocido como *Implicit Invocation*

Ejemplos?

MVC
Smalltalk

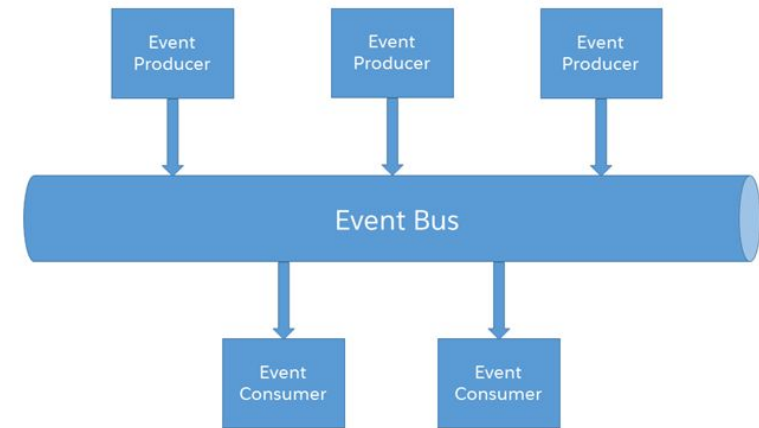
Publisher /
Subscriber

- Un componente puede publicar uno o más eventos (*broadcast*) en lugar de invocar otro componente directamente
- Otros componentes *registran* su interés en dicho evento
- El sistema invoca a *todos* los componentes registrados

Event Based Integration (EBI)

- Se reduce el acoplamiento, dado que se elimina la necesidad de identificación en la interfaz del conector
 - “Un componente no requiere conocer la identidad de los otros”
-

- Se requiere infraestructura, “el sistema”
- Usualmente EBI se implementa como un bus, donde los componentes escuchan por eventos de su interés



Event Based Integration (EBI)



Extensibilidad

Fácil agregar nuevos componentes que respondan a eventos existentes o nuevos



Reusabilidad

Establece un mecanismo de integración y una interfaz de eventos común



Evolucionabilidad

Permite reemplazar componentes sin afectar a otros



Escalabilidad

Número de notificaciones, tormenta de eventos, etc. Filtrado de eventos (capas), reduce simplicidad. Punto único de falla



Entendimiento

Difícil predecir qué ocurrirá ante un evento (ej. orden). Tampoco se sabe si alguien responderá



Eficiencia

En algunos sistemas, principalmente dominados por monitoreo, se elimina la necesidad de *polling*

Estilos Arquitectónicos #3

PEER TO PEER STYLES

1. Event Based Integration (EBI)
2. Distributed Objects (DO)
3. Brokered Distributed Objects (BDO)

DATA CENTERED ARCHITECTURES

4. Database
5. Blackboard

DATA FLOW STYLES

6. Pipe & Filter (PF)
7. Uniform Pipe & Filter (UPF)
8. Batch Sequential

Distributed Objects (DO)

- Estilo *Peer to Peer*

Organiza el sistema como un conjunto de componentes que interactúan como pares

- *Objeto como entidad abstracta* [...] que posee un *estado interno* (oculto, protegido) y operaciones asociadas [...]

Distributed Objects (DO)

- Una operación puede invocar otras operaciones, posiblemente en otros objetos [y otras, y otras]
 - Una cadena de invocaciones se conoce como 'acción'
-
- El estado se distribuye entre los objetos
-
- Para interactuar, los objetos deben conocer la identidad de los otros objetos
 - Cuando cambia la identidad de un objeto, se deben actualizar todos los objetos que explícitamente lo invoca
-
- Debe existir un controlador que gestione los objetos, interacción, etc.

Distributed Objects (DO)



Visibilidad

Difícil obtener una vista general del estado y de la actividad del sistema



Evolucionabilidad

Los objetos tiene interfaz pública, estado privado

Estilos Arquitectónicos #3

PEER TO PEER STYLES

1. Event Based Integration (EBI)
2. Distributed Objects (DO)
3. Brokered Distributed Objects (BDO)

DATA CENTERED ARCHITECTURES

4. Database
5. Blackboard

DATA FLOW STYLES

6. Pipe & Filter (PF)
7. Uniform Pipe & Filter (UPF)
8. Batch Sequential

Brokered Distributed Objects (BDO)

- Estilo *Peer to Peer*
- DO + LCS

Ej. CORBA

DO y BDO despertaron mucho interés, pero resultaron no ser muy eficientes frente a otros estilos basados en red

- Busca reducir el problema de la identidad en DO, colocando un estilo intermediador para facilitar la comunicación (ej. EBI, brokered CS)
- Los clientes utilizan nombres de servicio generales
- El broker resuelve la identidad del objeto a quien debe enviarse el request

Brokered Distributed Objects (BDO)



Reusabilidad

Uso de nombres genéricos



Evolucionabilidad

Uso de nombres genéricos



Eficiencia

Mayor cantidad de interacciones de red



Performance Perc.

Mayor cantidad de interacciones de red

Estilos Arquitectónicos #3

PEER TO PEER STYLES

1. Event Based Integration (EBI)
2. Distributed Objects (DO)
3. Brokered Distributed Objects (BDO)

DATA CENTERED ARCHITECTURES

4. Database
5. Blackboard

DATA FLOW STYLES

6. Pipe & Filter (PF)
7. Uniform Pipe & Filter (UPF)
8. Batch Sequential

Data Centered Architectures

Componentes

- Estructura central de datos, con el estado
 - Componentes independientes, que interactúan con dicho almacenamiento centralizado
-
- Si el *input stream* decide/dispara los procesos a ser ejecutados, el repositorio podría ser una **base de datos**
 - Si el estado actual es quien decide los procesos a ser ejecutados, el repositorio podría denominarse **blackboard**

Blackboard

Fuentes de conocimiento

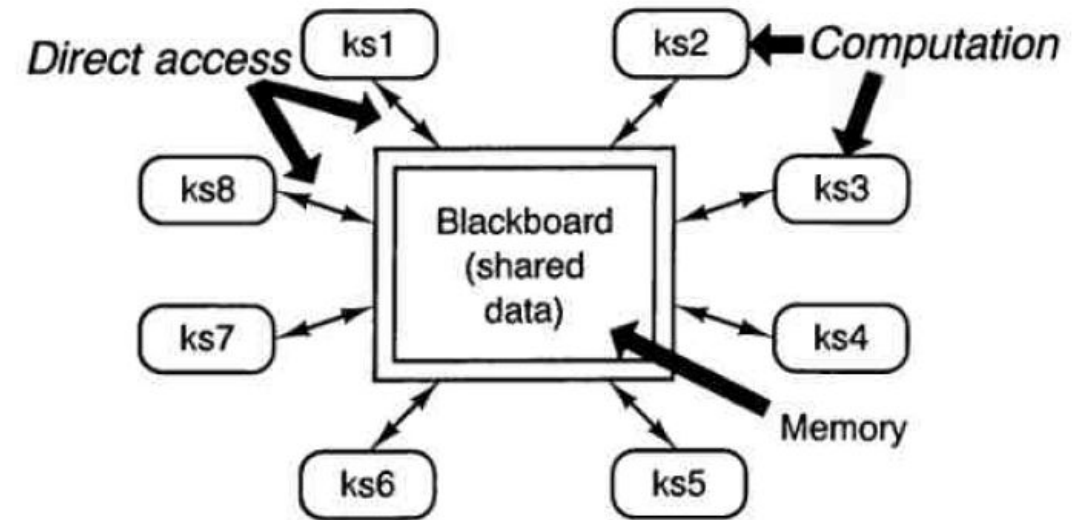
- Se comunican entre sí mediante el blackboard

Estructura de datos

- Datos para la solución del problema
- Modificado por las fuentes de conocimiento
- En forma iterativa se llega a la solución

Control

- Coordinado exclusivamente por el estado del blackboard
- Las fuentes de conocimiento responden oportunamente a los cambios del blackboard



Estilos Arquitectónicos #3

PEER TO PEER STYLES

1. Event Based Integration (EBI)
2. Distributed Objects (DO)
3. Brokered Distributed Objects (BDO)

DATA CENTERED ARCHITECTURES

4. Database
5. Blackboard

DATA FLOW STYLES

6. Pipe & Filter (PF)
7. Uniform Pipe & Filter (UPF)
8. Batch Sequential

Pipe & Filter (PF)

- Los componentes son los “*filtros*”
- Cada filtro lee *flujos* de datos (inputs), y produce *flujos* de datos (outputs)
- *Generalmente*, el flujo de entrada se transforma de forma *incremental* (La producción comienza antes que el input sea totalmente consumido)

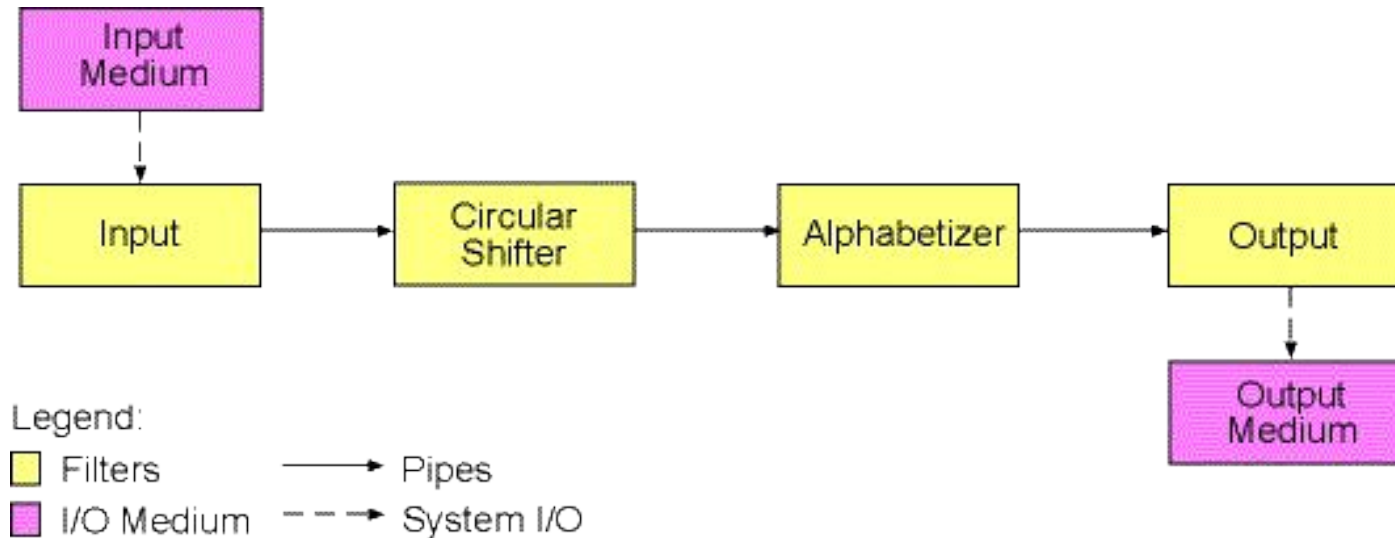
Restricción: cada filtro debe ser totalmente independiente de los otros

- Desacoplamiento total (los filtros se suponen programas independientes)
- No deben compartir estado (tampoco se conserva entre invocaciones)
- No deben conocer la identidad de otros filtros

Pipe & Filter (PF)

Se ve el sistema como una serie de transformaciones sucesivas

Útil si el problema responde al patrón de “corriente de flujo de datos”



Se podría medir el throughput del sistema

El *pipeline* podría ser una red, donde un output sea input de *más* de un filtro

Pipe & Filter (PF)



Simplicidad

Se puede entender el sistema como una composición simple del comportamiento de los filtros individuales



Reusabilidad

Los filtros pueden reusarse y conectarse entre sí, siempre que estén de acuerdo con sus interfaces



Extensibilidad

Nuevos filtros pueden agregarse



Evolucionabilidad

Los filtros pueden reemplazarse por una versión mejorada, sin impactar en los restantes



Configurabilidad

La aplicación podría determinar los filtros a utilizar



Verificabilidad

Permiten ciertos análisis (deadlocks, throughput, etc.)

Pipe & Filter (PF)

 Performance Perc.



Favorece el procesamiento concurrente



Largas tuberías generan retrasos de propagación



Podría verse afectada si el problema no se adapta al flujo de datos



Procesamiento batch si un filtro no puede procesar su entrada en forma incremental

Estilos Arquitectónicos #3

PEER TO PEER STYLES

1. Event Based Integration (EBI)
2. Distributed Objects (DO)
3. Brokered Distributed Objects (BDO)

DATA CENTERED ARCHITECTURES

4. Database
5. Blackboard

DATA FLOW STYLES

6. Pipe & Filter (PF)
7. Uniform Pipe & Filter (UPF)
8. Batch Sequential

Uniform Pipe & Filter (UPF)

- Deriva de PF
- Añade restricción: todos los filtros deben tener la misma interfaz
- Permite que filtros desarrollados independientemente puedan ser combinados a voluntad para formar nuevas aplicaciones

Ejemplo?


Unix

- Entrada = stdin
- Salida = stdout, stderr


Uniform Pipe & Filter (UPF)

 Simplicidad Más fácil entender cómo funciona un filtro (vs PF)

 Reusabilidad Mayores posibilidades que en PF

 Extensibilidad Mayores posibilidades que en PF

 Configurabilidad Mayores posibilidades que en PF

 Visibilidad Es más fácil para un componente monitorear la interacción entre dos filtros cualesquiera

 Performance Los datos podrían requerir ser convertidos desde/hacia el formato original/definido por la interfaz

Estilos Arquitectónicos #3

PEER TO PEER STYLES

1. Event Based Integration (EBI)
2. Distributed Objects (DO)
3. Brokered Distributed Objects (BDO)

DATA CENTERED ARCHITECTURES

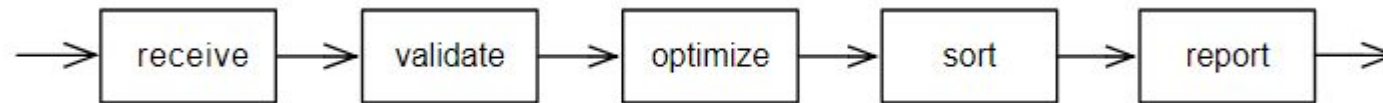
4. Database
5. Blackboard

DATA FLOW STYLES

6. Pipe & Filter (PF)
7. Uniform Pipe & Filter (UPF)
8. Batch Sequential

Batch Sequential

- Similar a PF o UPF
- Un filtro termina su procesamiento *antes* que comience el siguiente



- Simplifica la separación de un sistema en subsistemas
- Aumenta la latencia y disminuye el throughput
- Reduce las posibilidades de concurrencia

Estilos Arquitectónicos #3

PEER TO PEER STYLES

1. Event Based Integration (EBI)
2. Distributed Objects (DO)
3. Brokered Distributed Objects (BDO)

DATA CENTERED ARCHITECTURES

4. Database
5. Blackboard

DATA FLOW STYLES

6. Pipe & Filter (PF)
7. Uniform Pipe & Filter (UPF)
8. Batch Sequential

Estilos Arquitectónicos

- DATA FLOW STYLES
 - PIPE & FILTER
 - UNIFORM PIPE & FILTER
 - BATCH SEQUENTIAL
- REPLICATION STYLES
 - REPLICATED REPOSITORY
 - CACHE
- DATA CENTERED ARCHITECTURES
 - DATABASE
 - BLACKBOARD
- HIERARCHICAL STYLES
 - CLIENT-SERVER
 - LAYERED SYSTEMS
 - LAYERED CLIENT-SERVER
 - CLIENT-STATELESS-SERVER
 - CLIENT-CACHE-STATELESS-SERVER
 - LAYERED-CLIENT-CACHE-STATELESS-SERVER
 - REMOTE SESSION
 - REMOTE DATA ACCESS
- MOBILE CODE
 - VIRTUAL MACHINE
 - REMOTE EVALUATION
 - CODE ON DEMAND
 - LAYERED-CODE-ON-DEMAND-CLIENT-CACHE-STATELESS-SERVER
 - MOBILE AGENT
- PEER TO PEER STYLES
 - EVENT-BASED INTEGRATION
 - DISTRIBUTED OBJECTS
 - BROKED DISTRIBUTED OBJECTS

Resumen Fielding



Style	Derivation	Net Perform.	UP Perform.	Efficiency	Scalability	Simplicity	Evolvability	Extensibility	Customiz.	Configur.	Reusability	Visibility	Portability	Reliability
PF	PF		±			+	+	+		+	+			
UPF		-	±			++	+	+		++	++	+		
RR			++		+									+
\$	RR		+	+	+	+								
CS					+	+	+							
LS			-		+		+				+		+	
LCS	CS+LS		-		++	+	++				+		+	
CSS	CS	-			++	+	+					+		+
CSSS	CSS+\$	-	+	+	++	+	+					+		+
LCSSS	LCS+CSSS	-	±	+	+++	++	++				+	+	+	+
RS	CS			+	-	+	+					-		
RDA	CS			+	-	-						+		-
VM	CS+VM					±		+				-	+	
REV				+	-	±		+	+			-	+	-
COD			+	+	+	±		+		+		-		
LCODCSSS	LCSSS+COD	-	++	++	+4+	+±+	++	+		+	+	±	+	+
MA	REV+COD		+	++		±		++	+	+		-	+	
EBI				+	--	±	+	+		+	+	-		-
C2	EBI+LCS		-	+		+	++	+		+	++	±	+	±
DO	CS+CS	-		+			+	+		+	+	-		-
BDO	DO+LCS	-	-				++	+		+	++	-	+	

Table 3-6. Evaluation Summary

Consultas?

Feedback

<https://goo.gl/forms/NvrORS12kuuBitpE3>

Christian Calónico
ccalónico@fi.uba.ar