

# Arquitectura de Software

## Broker

**Broker (1)**

Arquitectura de Software

Rubén Béjar y José Merseguer



Universidad  
Zaragoza

1542

# Indice

- Objetivo
- Ejemplo
- Contexto
- Descripción del problema
- Solución del problema
- Estructura del patrón
- Dinámica del patrón
- Consecuencias de usar el patrón

# Objetivo

- El Broker es un patrón arquitectural
- Se utiliza para desarrollar sistemas software distribuidos
- Componentes desacoplados que interactúan por invocaciones de servicios remotos

# Responsabilidades

- El Broker lo vemos como un componente que:
  - Coordina la comunicación
  - Dirige las peticiones
  - Transmite resultados y excepciones

# Ejemplo

- Queremos desarrollar un sistema de información al ciudadano (CIS):
  - Tenemos servidores en la red que albergan servicios con información sobre restaurantes, hoteles, transporte público, etc.
  - Los turistas y los ciudadanos consultarán esta información a través de una aplicación cliente con sus portátiles, tabletas, etc.
  - El software (aplicación cliente) que construiremos recupera la información de los servidores

# Ejemplo ...

- Los servicios y la información cambian y crecen continuamente → Los servicios deben estar *desacoplados* entre sí
- La aplicación cliente accede a los servicios *sin conocer su ubicación* → Ello permite mover, duplicar y migrar los servicios

# Contexto

- El entorno es *distribuido* y *heterogéneo*:
  - Servicios implementados en diferentes lenguajes de programación
  - Servidores de diferentes tipos
  - Diferentes sistemas operativos

# Descripción del problema

- Construir un software complejo partiendo de componentes desacoplados. No estamos construyendo una aplicación monolítica
- El resultado debería ser que la aplicación gana en flexibilidad/extensibilidad, mantenibilidad y oportunidades de cambiar

# Problema ...

- Ventaja:
  - Cuando la funcionalidad está dividida entre componentes independientes → El sistema debería ser más fácil de escalar y de distribuir

# Problema ...

- Inconveniente:
  - ¿Deben los componentes manejar la comunicación?
    - No. Implicaría incorporar código extra
    - Los clientes serían dependientes del mecanismo de comunicación
    - Los clientes necesitarán conocer la localización de los servidores. Quizá un sólo lenguaje de programación

# Problema ...

- Necesidades:
  - Mecanismos para añadir, quitar, cambiar, localizar, activar servicios/componentes
  - Las aplicaciones que usen esos mecanismos no deben depender de tecnologías específicas ya que deben garantizar portabilidad e interoperabilidad

# Problema ...

- Idealmente para el desarrollador:
  - No debería haber diferencias entre desarrollar aplicaciones para sistemas centralizados distribuidos
  - Cuando una aplicación usa un objeto, sólo debe conocer su interfaz, no su implementación, ni su localización

# Problema ...

- ¿Cuándo usar el Broker?
  - Cuando se necesite cambiar, añadir o quitar componentes en tiempo de ejecución
  - Cuando la arquitectura deba esconder detalles del sistema, tecnología e implementación a quienes usen los componentes y sus servicios
  - Los clientes deben acceder a servicios proporcionados por otros (pero quizá dentro de la misma organización) a través de invocaciones remotas y transparentes con respecto a la localización

# Solución del problema

- Introducir un Broker para desacoplar clientes y servidores
- Los servidores se registran en el Broker y ofrecen sus servicios a través de interfaces
- Los clientes acceden a los servidores a través del Broker

# Solución ...

- Tareas del Broker
  - Registrar los servicios
  - Atender a los clientes
  - Localizar el servidor adecuado
  - Transmitir a los servidores las peticiones
  - Transmitir resultados y excepciones al cliente

# Solución ...

- Ventajas del Broker

- Las aplicaciones pueden acceder a servicios distribuidos sólo invocando métodos del objeto adecuado → Sin enfocarse en procesos de comunicación de bajo nivel
- Permite cambiar, añadir, quitar y recolocar objetos en tiempo de ejecución

# Solución ...

- Ventajas del Broker de objetos
  - Integran los sistemas distribuidos y la tecnología de objetos
  - Reduce la complejidad en el desarrollo de aplicaciones distribuidas:
    - La distribución es transparente al usuario
    - Los servicios distribuidos se encapsulan dentro de objetos
    - Extienden el modelo de objetos de una aplicación a una aplicación distribuida formada por componentes desacoplados que se ejecutan en entornos heterogéneos (distinto hw, distinto SO, diferentes LP)

# Estructura del patrón

- Componentes servidores
- Aplicaciones clientes
- Componente Broker
- Proxies del cliente
- Proxies del servidor
- Puentes (opcionales)

# Servidores

- Muestran su funcionalidad (servicios) a través de interfaces (atributos y métodos) -IDL o binarias-
- Se registran en los Brokers
- Diferentes tipos de componentes:
  - Servicios para dominios específicos
    - Ejemplos: servicios para geolocalización (mapas, ...), servicios para bioinformática, ...
  - Servicios generales, comunes a muchos dominios específicos
    - Ejemplo: servicios de pago por Internet

# Clients

- Son aplicaciones que acceden a los servicios de los componentes servidores
- Envían sus peticiones al Broker, no directamente a los servidores
- Reciben la respuesta (o excepción) del Broker no de los servidores
- No necesitan conocer la localización de los servidores → Permite añadir nuevos servicios o migrarlos (en tiempo de ejecución)

# Broker

- Es un mensajero. Responsable de retransmitir el mensaje del cliente al servidor y viceversa
- Sabe cómo localizar a los servidores, basado en un mecanismo de identificación única
- Ofrece un API con operaciones para que los servidores se (des)registren y para que los clientes invoquen los métodos
- Cuando le llega una petición la pasa al servidor correspondiente. Si está inactivo lo activará

# Broker ...

- Dos tipos de Broker:
  - Indirectos: el que estamos viendo
  - Directos: sólo establecen la comunicación inicial entre el cliente y el servidor, a partir de ahí el Broker se desentiende. Ver Client-Dispatcher-Server (323)

# Proxies del cliente

- Capa software entre el cliente y el Broker
- Median entre el cliente y el Broker
- Hacen que los servicios del Broker parezcan locales/*transparentes* al cliente; y viceversa, el Broker devuelve los resultados de manera transparente
- Esconde al cliente detalles de implementación:
  - Mecanismos de comunicación entre procesos (entre el cliente y el Broker)
  - Hacen el "marshaling" de los parámetros y envía la petición al Broker
  - Hacen el "unmarshaling" de los resultados y los envía al cliente

"marshaling": convertir datos en un formato independiente de la máquina –XDR, ASN-  
"unmarshaling": al revés

# Proxies del servidor

- Median entre el servidor y el Broker
- Esconden detalles de implementación:
  - Reciben las peticiones
  - Desempaquetan las peticiones
  - Hacen el "unmarshaling" de los parámetros
  - Llaman al servicio del servidor
  - Hacen el "marshaling" de los resultados y excepciones para enviarlos al Broker
- Hacen que los servicios del Broker parezcan locales/*transparentes* al servidor; y viceversa

# Puentes (opcionales)

- El servidor no está registrado en el Broker que recibe la petición
  - El Broker busca otro → *Interoperabilidad*
- Sirven para que interoperen los Brokers
- Los Brokers deben comunicarse independientemente del sistema operativo que usen y del tipo de red en la que residan → El puente esconde/encapsula esos detalles
- Media entre el Broker local y el puente del Broker remoto

# Diagrama de clases

# Dinámica del patrón

- Distinguimos tres escenarios:
  - Escenario I: Cómo se registra un servidor en un Broker
  - Escenario II: Cómo una aplicación cliente envía una petición (síncrona, asíncrona)
  - Escenario III: Cómo comunican los Brokers usando Puentes

# Escenario I

**Broker (28)**

Arquitectura de Software

Rubén Béjar y José Merseguer



**Universidad**  
Zaragoza  
1542

# Escenario II

**Broker (29)**

Arquitectura de Software

Rubén Béjar y José Merseguer



**Universidad**  
Zaragoza  
1542

# Escenario III

**Broker (30)**

Arquitectura de Software

Rubén Béjar y José Merseguer



**Universidad**  
Zaragoza  
1542

# Consecuencias del patrón

- Todo es extensible y reemplazable
  - Se puede cambiar la implementación pero no las interfaces/API de:
    - Servidores; sin afectar a los clientes
    - Broker; sin afectar a clientes ni servidores
  - Se puede cambiar los mecanismos de comunicación:
    - Cliente-broker, servidor-broker, broker-broker
    - Recomilar los clientes, servidores y brokers → crear nuevos proxies y puentes
  - No hay que cambiar el código fuente

Broker (31)

Arquitectura de Software

Rubén Béjar y José Merseguer



Universidad  
Zaragoza

1542

# Beneficios ...

- Transparencia en la localización

- Los clientes no necesitan saber dónde están ubicados los servidores (y viceversa)

- Interoperabilidad entre Brokers

- Pueden interoperar si utilizan el mismo protocolo para intercambiar mensajes. El protocolo lo implementan y gestionan los puentes, que traducen del protocolo específico del Broker al protocolo común

- Reusabilidad

- Podemos construir aplicaciones utilizando servicios

# Inconvenientes

- Peores prestaciones

- Disminuye la eficiencia con respecto a sistemas con componentes servidores estáticos y localizados. El Broker introduce niveles de indirección para ser portable, flexible e intercambiable

- Menor tolerancia a fallos

- que los sistemas no distribuidos. Si falla un broker o un servidor todas las aplicaciones que dependen de ellos no pueden seguir ejecutándose

# Beneficios/inconvenientes

- Pruebas y depuración:

- Hacer pruebas a un software construido con servicios que se están utilizando continuamente (ya están probados) es más fácil
- Depurar un Broker es un trabajo difícil y costoso