Tema 5

Patrones de arquitectura de sistemas

Ingeniería del Software Avanzada

Índice

[Esquema 3](#_Toc535213174)

[Ideas clave 4](#_Toc535213175)

[5.1. Introducción y objetivos 4](#_Toc535213176)

[5.2. El diseño arquitectónico 5](#_Toc535213177)

[5.3. Patrones arquitectónicos generales 7](#_Toc535213178)

[5.4. Arquitecturas tolerantes a fallos 15](#_Toc535213179)

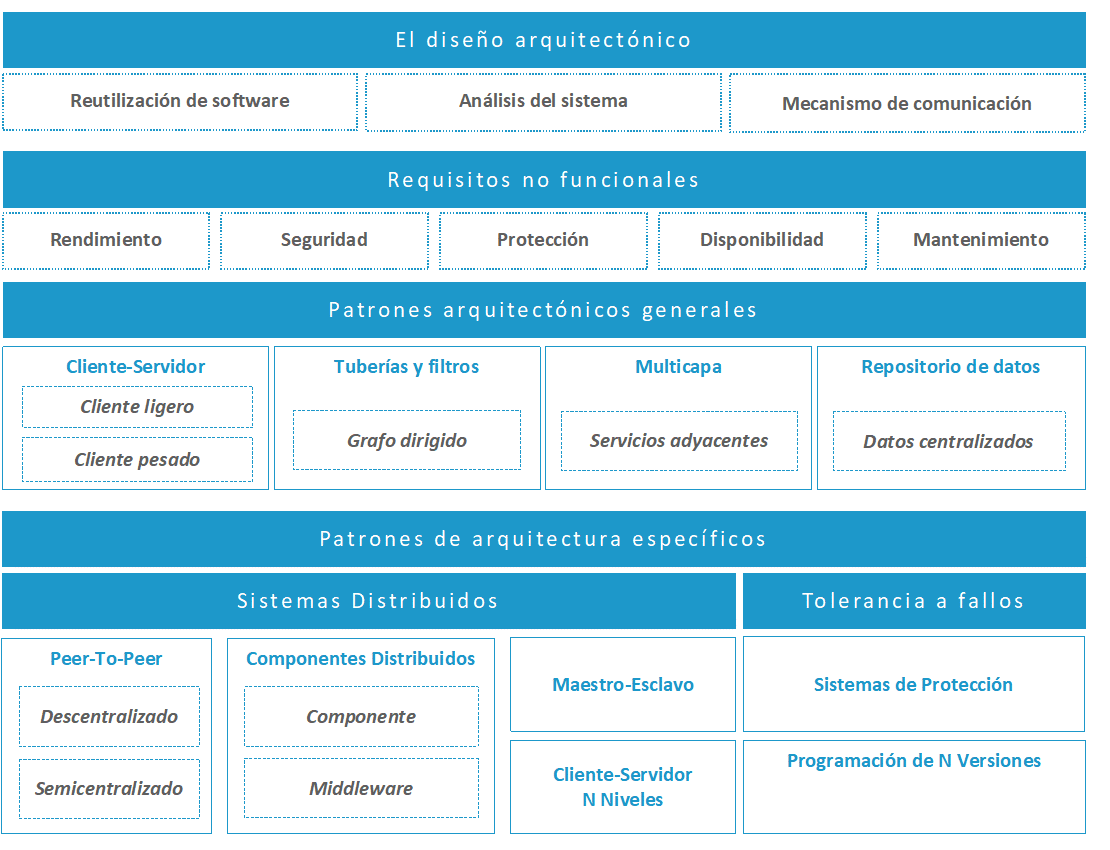
[5.5. Arquitecturas de sistemas distribuidos 18](#_Toc535213180)

[5.6. Referencias bibliográficas 26](#_Toc535213181)

[A fondo 27](#_Toc535213182)

[Test 29](#_Toc535213183)

Esquema



Ideas clave

5.1. Introducción y objetivos

Una de las técnicas de ingeniería que facilitan la **reutilización es el empleo de patrones para modelar la arquitectura general del sistema.** En este caso, no estamos utilizando funciones, componentes o servicios concretos, sino que estamos **aprovechando la experiencia y conocimiento acumulados** por otros desarrolladores en proyectos anteriores. Así, no sólo aceleramos el proceso de diseño, sino que aplicamos soluciones que están sobradamente probadas y comprobadas en la práctica. Al mismo tiempo, un correcto diseño puede ayudar a identificar ciertos elementos que es posible implementar reutilizando software existente, acelerando así también el proceso de implementación y pruebas.

Existen infinidad de situaciones que se repiten reiteradamente en sucesivos proyectos, y carece de sentido volver a pensar soluciones arquitectónicas para problemas que han sido resueltos en proyectos previos similares. En este caso, tal vez la mayor complejidad consiste en **identificar aquellas situaciones que presentan similitudes con otros problemas previamente analizados** y, en su caso, determinar la solución o soluciones más apropiadas partiendo de un patrón. En este contexto, **un patrón no es más que la descripción de una buena solución para un problema habitual en la práctica.**

En este tema pretendemos alcanzar los siguientes objetivos:

* **Comprender la importancia de los patrones de diseño arquitectónico y su utilidad** como herramienta de reutilización de conocimiento en la implementación de sistemas software.
* **Saber seleccionar una arquitectura** en función de la importancia de los diferentes requisitos no funcionales.
* **Conocer los patrones arquitectónicos generales** en los que se basan la mayoría de los sistemas informáticos: cliente–servidor, tuberías y filtros, multicapa y repositorio de datos.
* **Conocer algunos patrones de arquitectura específicos**, como los empleados para mejorar la tolerancia a fallos de un sistema o los empleados en sistemas distribuidos.

5.2. El diseño arquitectónico

**El diseño arquitectónico de un sistema de información es su representación esquemática al más alto nivel** (Polo Usaola, 2012). **Son modelos que prescinden de determinados detalles centrándose en aquellos elementos principales del sistema y en la misión que cumplen en el contexto global.**

El diseño arquitectónico de un sistema permite estudiarlo, y realizado adecuadamente ofrece las siguientes **ventajas**:

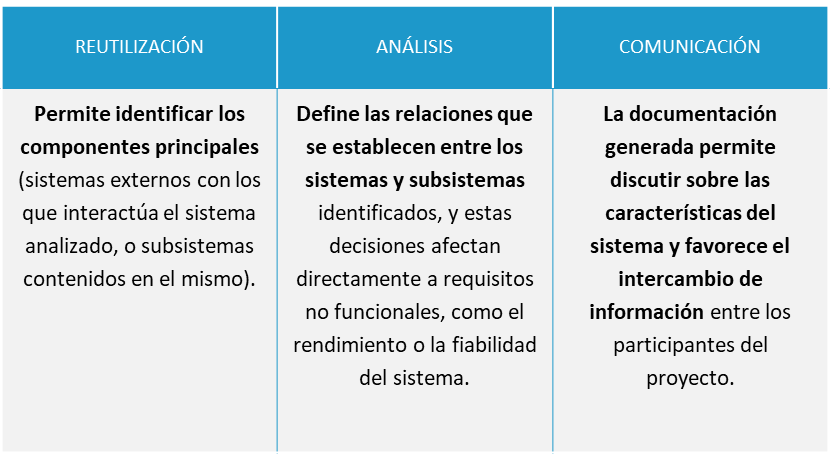


Figura 1. Ventajas del diseño arquitectónico de un sistema

La importancia de los requisitos no funcionales

A la hora de diseñar la arquitectura **es necesario reflexionar sobre las características del sistema**, y pensar tanto en las que comparte con otros sistemas de la misma categoría —que pueden permitir la aplicación de patrones concretos— como en las que le son específicas, especialmente en sus **requisitos no funcionales**:

* **Rendimiento**. El rendimiento de un sistema está relacionado con el nivel de aprovechamiento de los recursos disponibles. Una medida del rendimiento es el número de tareas realizadas en una unidad de tiempo o, recíprocamente, el tiempo medio que el sistema tarda en responder y completar una tarea.

Si el rendimiento es un factor crítico deberemos tener especial cuidado en la manera de distribuir el sistema. Será preferible concentrar las operaciones más críticas en un número reducido de nodos concentrados en pocas máquinas, puesto que los procesos de comunicación introducen retardos adicionales.

* **Seguridad.** Hace referencia a la capacidad de un sistema para protegerse ante intrusiones.

Si la seguridad y protección de los datos es fundamental, convendrá tenerlos agrupados en un único repositorio y definir una arquitectura de capas con varios niveles de seguridad que proteja el acceso a ellos.

* **Protección**. Este término hace referencia a la capacidad de un sistema para evitar o hacer frente a los fallos.

Si es importante que el sistema no falle (que sea fiable), o que en caso de fallo se recupere adecuadamente, será importante concentrar los procesos que realicen las operaciones más complejas y, posiblemente, definir mecanismos de redundancia que permitan mantener el sistema en funcionamiento a pesar de un fallo.

* **Disponibilidad.** Tiene que ver con la probabilidad, o fracción del tiempo total, en que el sistema proporciona los servicios cuando le son solicitados. Los sistemas de alta disponibilidad son a menudo redundantes, de manera que ante el fallo de un elemento otro pueda entrar a sustituirlo. Así también es posible realizar operaciones de sustitución o mantenimiento sin detener el sistema.
* **Mantenibilidad.** Este requisito es favorecido por el empleo de componentes de granularidad pequeña, fácilmente actualizables o sustituibles, y estableciendo una clara separación entre productores y consumidores de datos. En definitiva, un bajo acoplamiento facilita el mantenimiento del sistema.

Como vemos, hay requisitos que pueden estar en conflicto sugiriendo estilos arquitectónicos en contraposición. En muchas ocasiones, será necesario tomar decisiones de compromiso en función de la importancia relativa de los requisitos no funcionales.

5.3. Patrones arquitectónicos generales

Para Sommerville (2011), los patrones arquitectónicos son una manera de «representar, compartir y reutilizar el conocimiento sobre los sistemas de software» que se manifiesta como una «descripción abstracta estilizada de buena práctica, que se ensayó y puso a prueba en diferentes sistemas y entornos» (p. 156). Es decir, **un patrón ha debido mostrar su utilidad en desarrollos previos, e idealmente debería ir acompañado de recomendaciones sobre su aplicabilidad, fortalezas y debilidades.**

Arquitecturas cliente–servidor

**En estos sistemas la lógica se distribuye entre dos tipos de máquinas, clientes y servidores, conectados por algún tipo de *middleware* (Polo Usaola, 2012).**

Ambos tipos de sistemas pueden tener arquitecturas internas muy diferentes. Estas arquitecturas son habituales en los sistemas distribuidos, pero también puede aplicarse a procesos que se ejecutan en la misma máquina.

Los elementos fundamentales son:

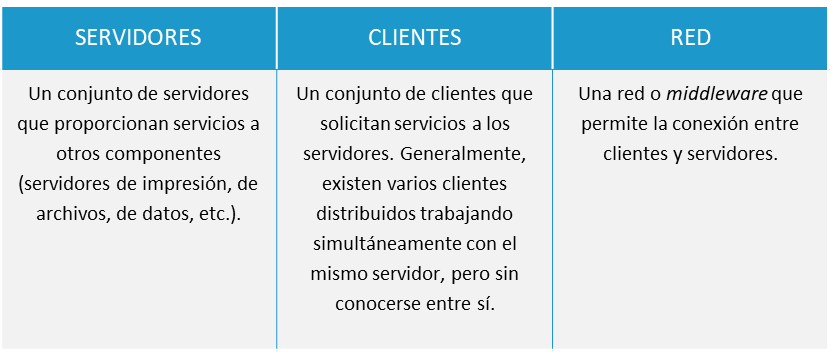


Figura 2. Elementos fundamentales de arquitecturas cliente–servidor

En la figura 3 se muestra un ejemplo de un **museo virtual como sistema multiusuario**, donde además se aprecia que los distintos servidores a los que acceden los clientes se comportan a su vez como clientes frente a sus respectivos repositorios de datos, definiéndose un **modelo de dos capas en cada uno de los servidores**.

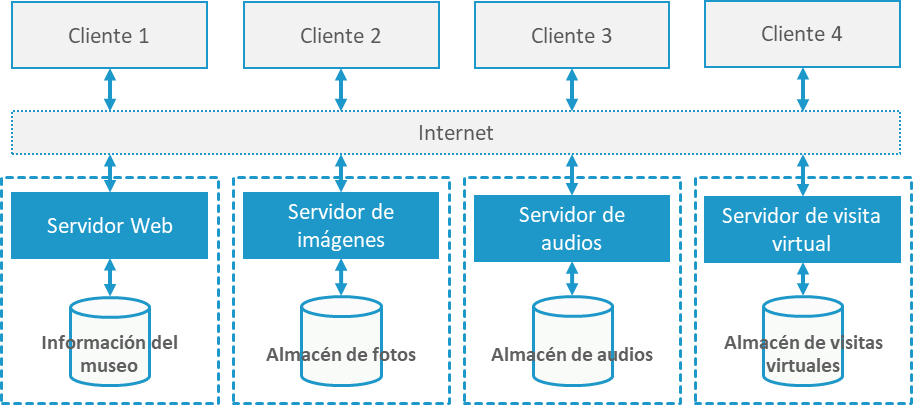


Figura 3. Ejemplo de patrón cliente–servidor en un museo virtual

La tabla 1 resume las **principales características de este patrón arquitectónico**. Esta es una manera habitual, aunque resumida, de describir los patrones; al menos se espera una descripción, algún tipo de ejemplo práctico, recomendaciones sobre su aplicación y alguna valoración crítica, indicando las principales ventajas e inconvenientes de su empleo.

|  |  |
| --- | --- |
| **NOMBRE** | **PATRÓN CLIENTE-SERVIDOR** |
| **Descripción** | La funcionalidad del sistema se organiza en servicios que ofrecen distintos servidores. Los clientes son usuarios de estos servicios. |
| **Ejemplo** | En la figura 3 se muestra un ejemplo para un museo virtual organizado según el modelo cliente–servidor. |
| **Cuando** **usarlo** | Apropiado cuando desde diferentes puntos se necesita obtener información o servicios comunes. |
| **Ventajas** | * Los servidores pueden estar distribuidos en una red. * Cuando la carga es variable se pueden replicar los servidores. |
| **Desventajas** | * Los servidores son puntos vulnerables al poder recibir ataques. * La carga puede ser impredecible cuando aumenta el número de clientes. |

Tabla 1. Descripción del patrón cliente–servidor

En función de cómo esté distribuida la lógica entre los diferentes nodos, podemos encontrar diferentes situaciones particulares, como se aprecia en la tabla 2. Interesa resaltar la diferencia entre:

* **«Clientes ligeros»**, que únicamente contienen lógica de presentación.
* **«Clientes pesados»**, que, además, realizan operaciones adicionales con los datos y, en muchos casos, almacenan también información.

|  |  |
| --- | --- |
| **MODELO ARQUITECTÓNICO DETALLADO** | **EJEMPLO DE SISTEMA** |
|  | Sistema de presentación remota  (cliente ligero)  **Ejemplo:** telnet. |
|  | Sistema de presentación distribuida (cliente ligero)  **Ejemplo:** Páginas ASP, JSP, etc. |
|  | Sistemas de datos distribuidos  (cliente pesado)  **Ejemplo:** Sistema de gestión de una empresa con varias sucursales. |
|  | Sistema de lógica distribuida  (cliente pesado)  **Ejemplo:** Un simulador de préstamos. |
|  | Sistema remoto de gestión de datos (cliente pesado)  **Ejemplo:** Un cliente de base de datos. |

Tabla 2. Posibles repartos de responsabilidades en arquitecturas cliente–servidor. Fuente: Elaboración propia a partir de Polo Usaola (2012, p. 36)

Arquitecturas de tuberías y filtros

**Las arquitecturas de «tuberías y filtros» (*pipes and filters*) modelan sistemas cuyos componentes principales son nodos especializados en aceptar datos de entrada, procesarlos y producir unos resultados con ellos.**

Estos nodos, o filtros, se interconectan entre sí a través de algún mecanismo de comunicación (las tuberías), de manera que la representación global del sistema adopta la forma de un **grafo dirigido**.

Una característica importante de estos sistemas es que los **nodos son independientes entre sí**, no comparten información de estado ni tienen por qué conocer al resto de nodos de la red.

Un ejemplo clásico de este tipo de sistema son los **compiladores de lenguaje**s y, en general, cualquier sistema de procesamiento de lenguaje o de señal, como se muestra en la figura 4. Ejemplo de patrón tuberías y filtros en un esquema simplificado de compilación y enlace.

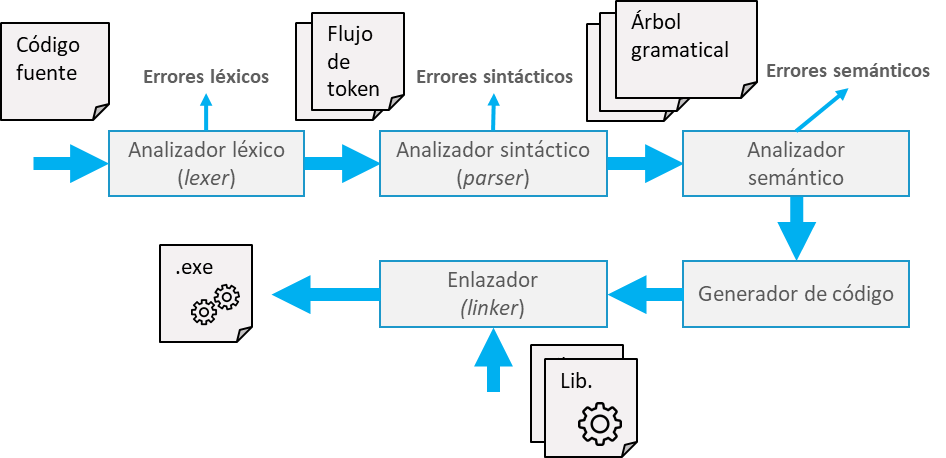


Figura 4. Ejemplo de patrón «tuberías y filtros» en un esquema simplificado de compilación y enlace

En este tipo de sistemas es fácil la reutilización de componentes, o filtros, siempre y cuando los protocolos de comunicación entre ellos sean respetados. Además, dada la independencia de los componentes el mantenimiento suele ser sencillo.

Las **características principales de este patrón** se resumen en la tabla 3.

|  |  |
| --- | --- |
| **NOMBRE** | **PATRÓN «TUBERÍAS Y FILTROS» (*PIPES AND FILTERS*)** |
| **Descripción** | Los datos fluyen a lo largo del sistema atravesando componentes de procesamiento (filtros) conectados entre sí y que no comparten estado. |
| **Ejemplo** | En la figura 4 se muestra un ejemplo con los diferentes y consecutivos procesamientos que sufre el código durante la compilación y enlace. |
| **Cuando usarlo** | Es de utilidad en cualquier operación de tratamiento de datos (en lote o transacciones) donde son necesarias varias etapas de procesado. |
| **Ventajas** | * Fácil de entender, mantener y aplicar reutilización. * Se adapta bien a muchos procesos empresariales. * Fácil extensión mediante la adición de más filtros. |
| **Desventajas** | * Se deben respetar formatos e interfaces entre filtros. * Puede requerir el desarrollo de filtros adicionales de acoplamiento. |

Tabla 3. Descripción del patrón «tuberías y filtros»

Arquitecturas multicapa

**En este caso, el sistema es susceptible de ser descompuesto en diferentes capas (*tiers* o *layers*, en inglés), de manera que cada una de ellas se especializa en un conjunto de responsabilidades y ofrece servicios a las capas situadas en el nivel inmediatamente superior.**

Idealmente, cada capa solo conoce a las adyacentes y solo ofrece servicios a la que está situada inmediatamente por encima, pero pueden encontrarse variaciones de este modelo.

Este tipo de arquitectura soporta el **enfoque de modelo incremental de sistemas**, comenzando por el desarrollo de las capas inferiores, de manera que sus servicios estén disponibles cuando comienzan a desarrollarse las superiores. También es apropiado cuando se necesita establecer distintos niveles de control de acceso a un sistema. Los sistemas operativos siguen también este patrón arquitectónico.

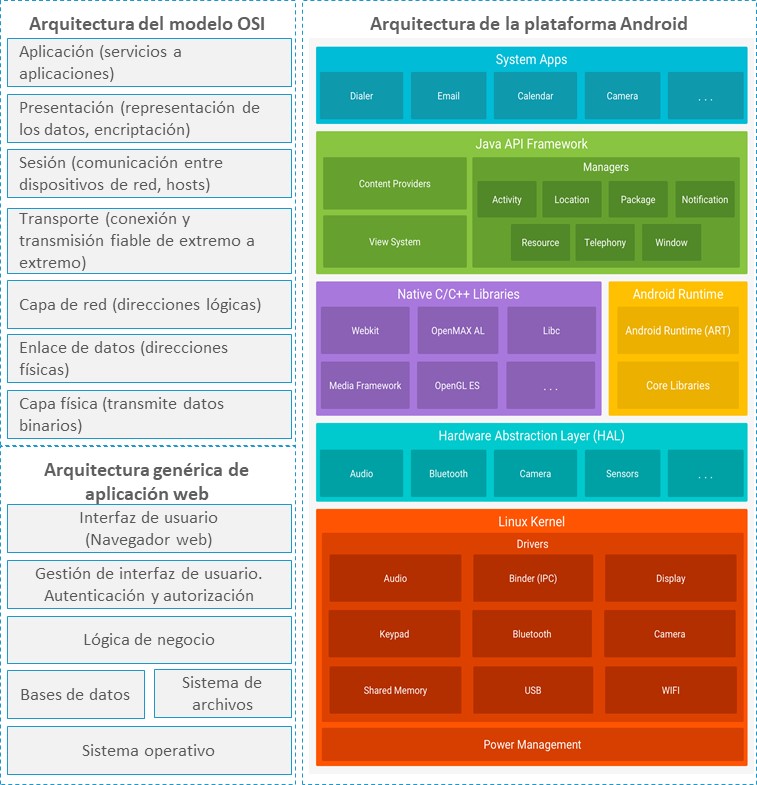


Figura 5. Tres ejemplos de arquitectura de capas. La arquitectura de comunicación del modelo OSI, un modelo genérico para aplicaciones web, y la arquitectura de capas de la plataforma Android. Fuente de la imagen de arquitectura Android: (Android, 2018)

En la arquitectura de capas de la plataforma Android, figura 5, encontramos un Kernel Linux en el nivel inferior, y sobre él se van construyendo servicios hasta llegar a la capa de API, que utilizan los programadores de aplicaciones.

|  |  |
| --- | --- |
| **NOMBRE** | **PATRÓN DE ARQUITECTURA EN CAPAS** |
| **Descripción** | Las funcionalidades del sistema se distribuyen en distintas capas lógicas, de manera que cada una ofrece servicios a la superior. |
| **Ejemplo** | En la figura 5 se presentan tres ejemplos de arquitectura, una para el caso de una aplicación, otra para el caso de un sistema de comunicación y finalmente la arquitectura de la plataforma Android. |
| **Cuando usarlo** | Cuando queramos construir nuevos servicios sobre otros existentes, o exista un requisito de seguridad en varios niveles. |
| **Ventajas** | * Permite la sustitución de capas enteras manteniendo la interfaz. * Permite el despliegue de cada capa en máquinas diferentes. |
| **Desventajas** | * Puede ser difícil mantener una separación clara entre capas. * Puede disminuir el rendimiento en sistemas con muchas capas. |

Tabla 4. Descripción del patrón de arquitectura en capas

Arquitectura de repositorio

**Este tipo de arquitectura es adecuada para sistemas que utilizan grandes cantidades de información que comparten diferentes subsistemas. Generalmente existen componentes generadores de datos y otros que los consumen, de manera que los datos quedan almacenados en un repositorio común.**

En la figura 6 se muestra un **ejemplo de este patrón para el caso de un sistema de gestión hospitalaria**. Todos los subsistemas comparten información sobre los pacientes, que está almacenada y centralizada en un único servidor. En este servidor se puede definir una **arquitectura interna orientada a la seguridad** como una arquitectura de capas. Este modelo no presenta diferencias respecto del cliente–servidor, pero está orientado a la concentración de los datos en el repositorio común.

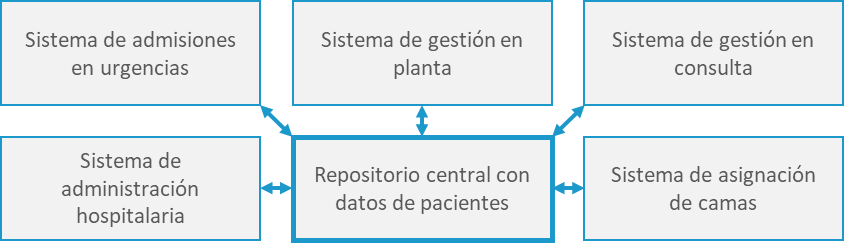


Figura 6. Ejemplo de una arquitectura de repositorio para un sistema de gestión hospitalaria

Esta es una **manera eficiente de compartir grandes cantidades de datos**, pero todos los participantes deben ponerse de acuerdo en un **modelo de datos común**.

El repositorio se encuentra lógicamente centralizado, aunque a nivel físico pueda estar distribuido en diferentes máquinas. Esta posible distribución, orientada en general a facilitar la escalabilidad del sistema, es uno de los puntos que plantea problemas en este modelo. **Lógicamente, al tener los datos centralizados en un único lugar, este es precisamente el punto más vulnerable del sistema.**

|  |  |
| --- | --- |
| **NOMBRE** | **ARQUITECTURA DE REPOSITORIO** |
| **Descripción** | Todos los datos del sistema se concentran en un repositorio central al que acceden el resto de las componentes. |
| **Ejemplo** | En la figura 6 vemos un ejemplo de un sistema de gestión hospitalaria, donde toda la información sobre pacientes es compartida por otros subsistemas de gestión. |
| **Cuando usarlo** | Con sistemas que deben almacenar grandes volúmenes de información durante mucho tiempo. |
| **Ventajas** | * El resto de las componentes son independientes entre sí. * Los datos están concentrados, lo cual facilita su gestión. |
| **Desventajas** | * La información está concentrada en un único nodo crítico. * El resto de las componentes se comunican a través de este nodo. * Puede ser complicado distribuir el repositorio en varias máquinas (problemas de redundancia o inconsistencia de datos). |

Tabla 5. Descripción del patrón de arquitectura de repositorio

5.4. Arquitecturas tolerantes a fallos

La **confiabilidad de un sistema** es, según Sommerville (2011), **«una propiedad agregada que toma en cuenta la protección, fiabilidad, disponibilidad, seguridad y otros atributos del sistema, y refleja el grado en el que los usuarios pueden confiar en él» (p. 735).**

Sistemas críticos —como los de control de proceso, sistemas médicos o sistemas de control de vuelo— requieren de técnicas específicas para mejorar su confiablidad y entre ellas están las que refuerzan la arquitectura del sistema. Esto pasa por **analizar cuidadosamente la tolerancia a fallos** incluyendo, en general, componentes y mecanismos redundantes. Vamos a ver aquí un par de ejemplos de estas arquitecturas.

Sistemas de protección

**Un sistema de protección está especializado en monitorizar el correcto funcionamiento de otro sistema al que está asociado.** Son habituales en los sistemas de control de proceso (como la industria química) o control de equipo (como los procesos de fabricación o el control de vehículos).

El sistema de protección funciona en paralelo con el controlador de proceso, monitorizando tanto las condiciones ambientales como el estado del propio proceso. En caso de detectar alguna desviación sobre el comportamiento esperado, puede tomar el control del sistema o, al menos, provocar algún tipo de aviso al exterior. En la figura 7 podemos observar un ejemplo de ello.

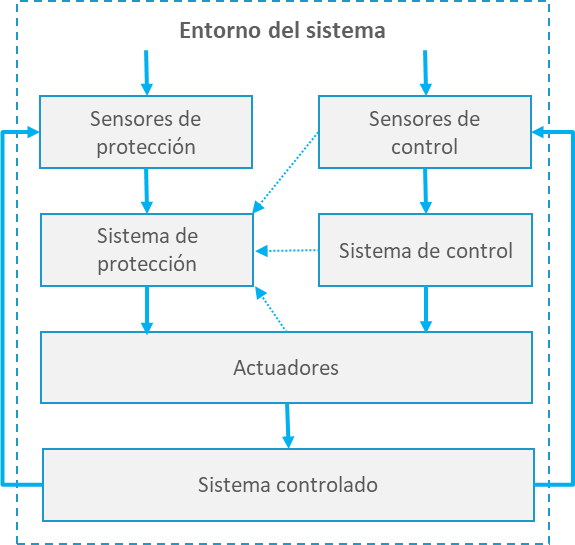


Figura 7. Ejemplo de un sistema de protección

**El sistema de protección adquiere información del entorno a través de un conjunto independiente de sensores.** También monitoriza otras variables del sistema, como el estado de los sensores y actuadores de proceso, y el propio sistema de control. Puede contar con un sistema de actuadores independiente al del proceso (por ejemplo, un sistema de detención de emergencia en un vehículo no tripulado). En caso de ser necesario, puede entrar en acción y evitar un fallo catastrófico del sistema.

Programación de varias versiones

**En este caso, se desarrollan varias versiones del mismo sistema que se ejecutan en máquinas diferentes. Partiendo de una especificación común, las distintas versiones son desarrolladas por diferentes equipos, y así se llega a implementaciones alternativas.**

Las salidas que generan los diferentes sistemas se comparan en un sistema de votación, rechazando aquellas salidas que son inconsistentes o que no se producen a tiempo. Este tipo de protección es habitual en los **sistemas de señalización ferroviaria, sistemas de control de aeronaves y protección de reactores** (Sommerville, 2011, p. 353).

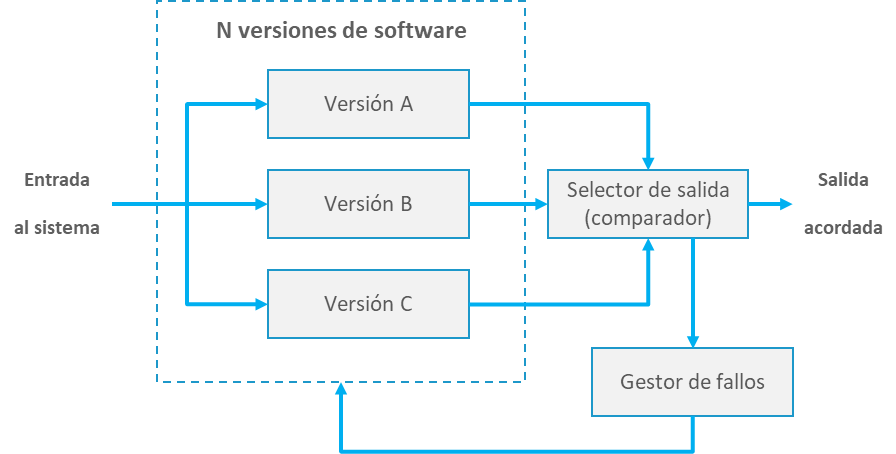


Figura 8. Ejemplo de protección con 3 versiones de software

5.5. Arquitecturas de sistemas distribuidos

Actualmente, la mayoría de grandes sistemas son sistemas distribuidos, donde el cómputo se reparte entre un gran número de máquinas y se contrastan con los sistemas centralizados cuyos componentes se ejecutan en un solo computador.

**Para Tanenbaum y Steen (2007), un sistema distribuido es una colección de computadores independientes que aparecen al usuario como un solo sistema coherente.**

Según Sommerville (2011), estos sistemas ofrecen múltiples ventajas (p. 480):

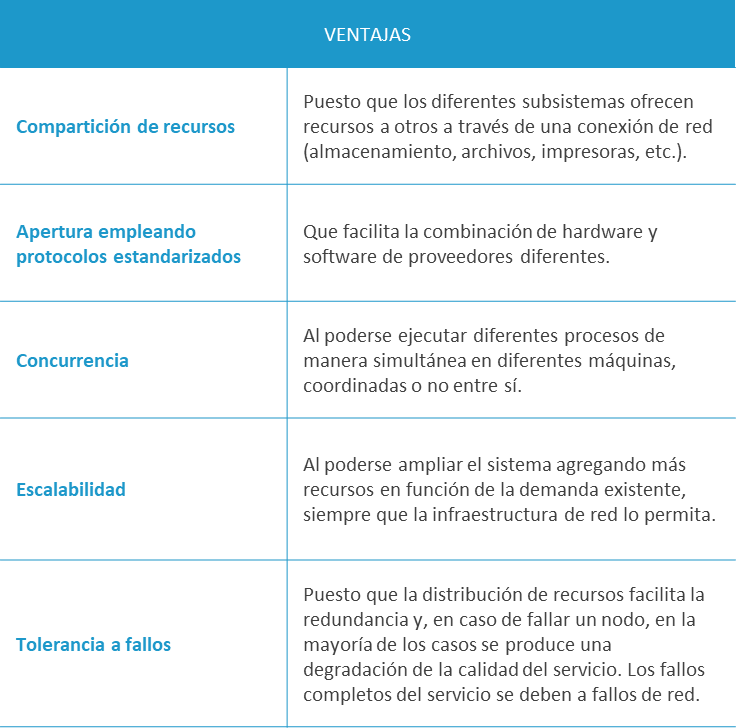


Figura 9. Ventajas de los sistemas distribuidos

Estos sistemas son más complejos que los centralizados, tanto en su diseño como en su implementación. En general, según Sommerville (2011), se debe encontrar un equilibrio entre diferentes requisitos no funcionales —como el rendimiento, la confiabilidad, la seguridad y la manejabilidad del sistema (p. 490)—. Con el tiempo han surgido algunos modelos arquitectónicos que se ocupan de manera específica de este tipo de problemas.

Arquitecturas maestro–esclavo

**Existen muchos sistemas con fuertes restricciones de tiempo real, en los que el cómputo se distribuye a diferentes escalas entre diferentes nodos de procesamiento.** Puede haber nodos especializados en la adquisición de información del entorno, nodos especializados en controlar los actuadores, nodos especializados en cómputo, etc.

En este tipo de sistemas es habitual la presencia de un proceso «maestro», responsable de gestionar la comunicación y coordinar al resto de participantes. Se trata de una arquitectura centralizada en la que el maestro toma la forma de un nodo cliente–servidor; actúa como cliente frente a otros procesos, como el control de sensores (pidiendo información) o el control de actuadores (enviando consignas de operación), y sirve información a otros procesos, como los clientes de las consolas de operador.

En la figura 10 se muestra un ejemplo de esta arquitectura, correspondiente a un sistema de gestión de tráfico.

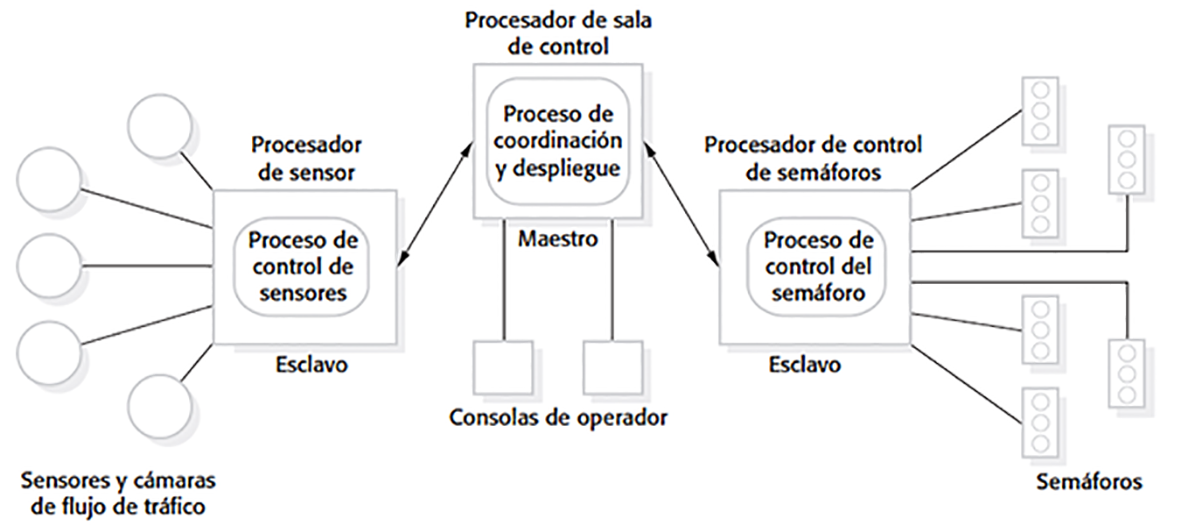


Figura 10. Ejemplo de arquitectura maestro–esclavo: sistema de control de tráfico.  
Fuente: Sommerville (2011, p. 491)

Arquitecturas cliente–servidor de varios niveles

El modelo más simple de arquitectura cliente-servidor fue presentado al comienzo del tema. **El patrón cliente-servidor es predominante en sistemas distribuidos y, en el caso más básico, presenta únicamente dos niveles lógicos: existe un único servidor y un conjunto más grande de clientes que consumen sus servicios.**

Este modelo es compatible con una arquitectura multicapa, pero presenta el **problema** de que todas las capas (presentación procesamiento de aplicación, gestión de datos y base de datos) se mapean en dos únicos sistemas cómputo. Ello perjudica el rendimiento y la escalabilidad del sistema, especialmente cuando elegimos un modelo de cliente ligero.

**Las arquitecturas cliente–servidor de varios niveles son aquellas que distribuyen los diferentes servidores, con capacidades específicas, en diferentes niveles físicos.** En la tabla 6 se resumen las principales aplicaciones de cada modelo.

|  |  |
| --- | --- |
| **ARQUITECTURA CLIENTE-SERVIDOR** | **APLICACIONES** |
| **Dos niveles**  **(cliente ligero)** | * Sistemas heredados donde no es posible separar la lógica de negocio de la gestión de datos. * Aplicaciones de cómputo intensivo con poca gestión de datos. * Aplicaciones intensivas en datos con poco procesamiento. |
| **Dos niveles**  **(cliente pesado)** | * El procesamiento lo realiza software comercial en el cliente. * Se requiere procesamiento intensivo en el cliente. * No se garantiza la conexión a la red, y puede ser necesario realizar procesamiento en el cliente con datos almacenados en caché. |
| **Multinivel** | * Aplicaciones muy grandes con miles de usuarios. * Aplicaciones que manejan datos de múltiples fuentes. |

Tabla 6. Variantes del patrón cliente–servidor en sistemas distribuidos.  
Fuente: Elaboración propia a partir de Sommerville (2011, p. 495)

Ejemplos de aplicaciones apropiadas para emplear un **cliente ligero** son los procesos de compilación en lote (intensivos en cómputo) o la navegación web (intensivo en datos). En el caso de los **clientes pesados**, encontraríamos aplicaciones como la visualización de datos (intensiva en cómputo en el lado del cliente).

Arquitecturas de componentes distribuidos

La arquitectura multicapa, entendida en este contexto como una arquitectura cliente–servidor multinivel, permite que cada una de ellas se implemente como un **servidor separado**. Esto puede **plantear problemas de diseño**; no siempre está claro qué servicios incluir en cada capa, y puede plantear **problemas de escalabilidad** a la hora de replicar los servidores de cada capa a medida que se agregan clientes.

Un enfoque más flexible consiste en diseñar el sistema como un conjunto de servicios comunicados a través de un *middleware*, como se muestra en la figura 7. Aquí se ha adaptado el ejemplo anterior de la Figura 1 que se enfocaba en el patrón cliente-servidor de dos niveles, con cada servidor modelado en una arquitectura de dos capas. La nueva arquitectura de componentes distribuidos resulta más flexible, y parece más sencillo poder agregar nuevas bases de datos al sistema.

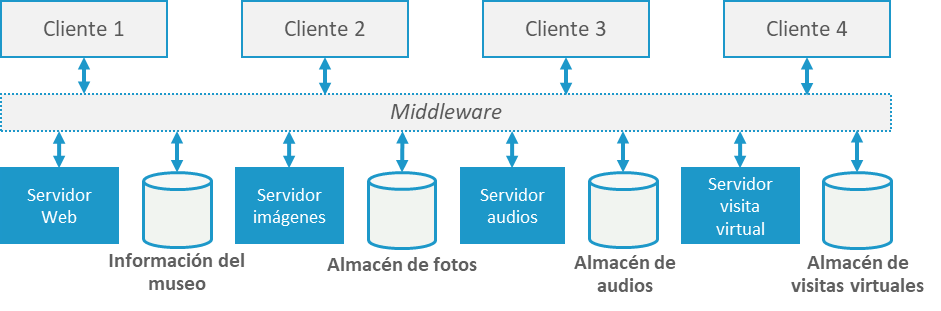


Figura 11. Ejemplo de arquitectura de componentes distribuidos para un museo virtual

En este patrón identificamos una serie de beneficios y desventajas:

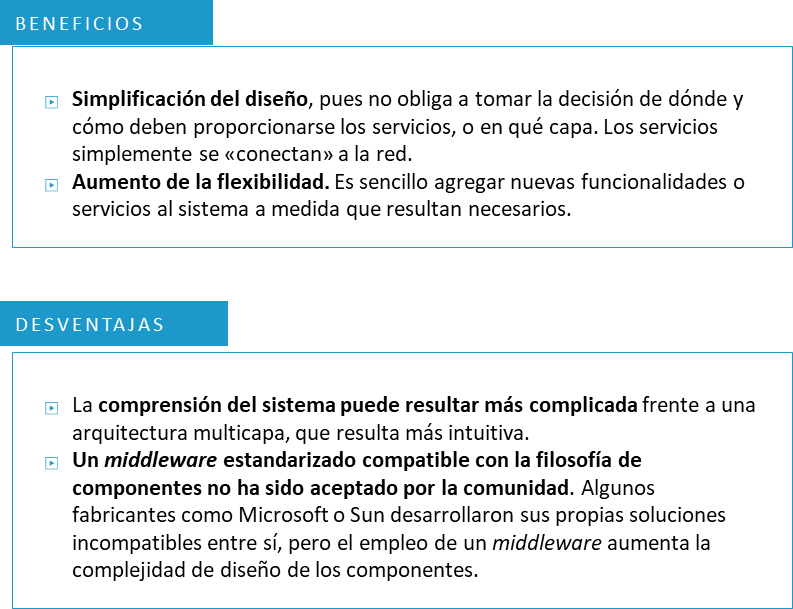


Figura 12. Beneficios y desventajas de las arquitecturas de componentes distribuidos

**Por estos motivos las arquitecturas de componentes distribuidos son a menudo sustituidas por arquitecturas orientadas a servicios.**

**CORBA (*Common Object Request Broker Architecture*)**

La Arquitectura Común de Intermediario de Peticiones de Objetos fue propuesta por el *Object Management Group* en la década de 1990 como *middleware* que permitía la comunicación entre componentes distribuidos.

Aparecieron varias implementaciones, pero nunca llegó a ser plenamente aceptado. Pronto fue sustituido por tecnologías propietarias (como Enterprise Java Beans o .NET) o reemplazado por arquitecturas orientadas a servicios.

En la siguiente web encontrarás más información sobre esta tecnología de middleware.

Accede a la página web a través del aula virtual o desde la siguiente dirección:

<http://www.SoftwareEngineering-9.com/Web/DistribSys/Corba.html>

Arquitecturas P2P

Las **arquitecturas P2P (*peer-to-peer*)**, o entre pares, contienen varios sistemas que se intercomunican entre sí en condiciones de igualdad. En este caso, **todos los elementos del modelo son homogéneos**, tienen características y funcionalidades similares. Esto no quiere decir que sean idénticos en su implementación interna, pero sí que al menos **desde el punto de vista externo ofrecen las mismas interfaces; tienen las mismas necesidades, y ofrecen los mismos servicios.**

Dicho con otras palabras, todos los nodos son simultáneamente clientes y servidores para el resto, y las interfaces son idénticas. Esta interfaz común es susceptible de ser implementada a través de algún componente reutilizable, que simplemente se integran en la implementación de los distintos componentes concretos del sistema P2P.

La **ventaja de este enfoque es que permite aprovechar mejor los recursos computacionales disponibles en la red**, al no hacer una distinción clara entre nodos servidores y nodos cliente. Esta tecnología se ha venido utilizando más frecuentemente en el ámbito personal que en el empresarial, en forma de **sistemas para compartir archivos basados en protocolos como Gnutella o BitTorrent, o sistemas de mensajería como Skype.**

A nivel empresarial e institucional también se utilizan en aplicaciones que requieren un cómputo intensivo, como alternativa a la compra de un único sistema hardware de alto rendimiento. Se distribuye así el procesamiento entre infinidad de nodos equivalentes, que aprovechan las capacidades locales de las máquinas de los usuarios de la red. Ejemplo de ello es el proyecto **SETI@home** (*University of California*, 2018), que pretende encontrar señales de inteligencia extraterrestre (*Search for Extra Terrestrial Intelligence*) analizando de manera distribuida datos adquiridos por radio telescopios.

Sin embargo, estos sistemas deben ofrecer algún mecanismo que permita a los nodos descubrirse y conectarse entre sí. En un esquema P2P puramente descentralizado, todos los nodos deberían comportarse como conmutadores de comunicaciones, actuando de intermediarios en los procesos de conexión.

Supongamos que la **figura 13** muestra un **sistema P2P de compartición de archivos**. En un momento dado el usuario del Nodo1 solicita un documento que solo está disponible en el Nodo8. Solo puede enviar la solicitud de búsqueda a sus nodos cercanos, el 2 y el 3, pero al no existir el documento en estos lugares, la solicitud se propaga hasta que el documento se encuentra en un nodo. Así, al menos, los nodos 3, 5, 6, 7, y 9 deberían participar en la búsqueda hasta que el documento es hallado en el Nodo8. En este momento, el Nodo8 ya puede establecer una conexión directa punto a punto con el Nodo1 y enviarle el documento.

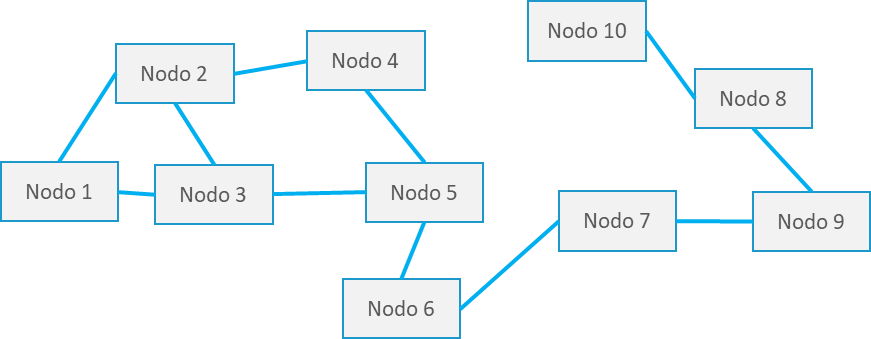


Figura 13. Ejemplo de arquitectura P2P descentralizada

La alternativa es diseñar un **sistema P2P semicentralizado**, donde dentro de la red algunos nodos actúan como servidores para el resto facilitando las comunicaciones y reduciendo el volumen de tráfico entre nodos. Estos nodos (o superpares) actúan como centralitas, permitiendo a los nodos descubrir cuáles otros están disponibles. También pueden ejercer otras actividades de coordinación y supervisión dentro de la red.

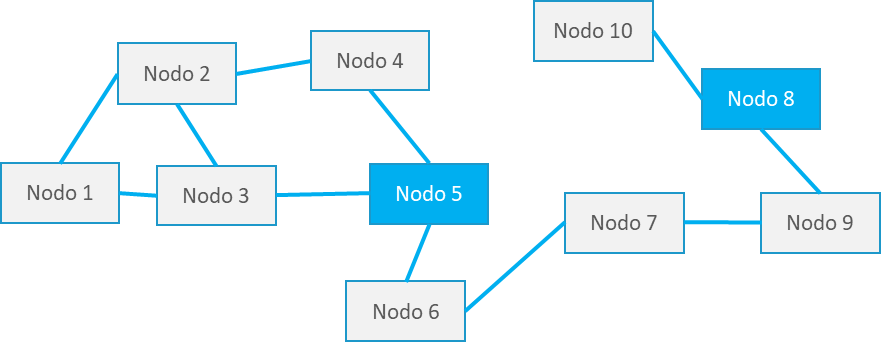


Figura 14. Ejemplo de arquitectura P2P semicentralizada, con dos «superpares»

5.6. Referencias bibliográficas

Android. (2018). Arquitectura de la plataforma [Web]. Recuperado de <https://developer.android.com/guide/platform/?hl=es-419>

Polo, M. (2012). *Desarrollo de software basado en reutilización* [Material didáctico]. Barcelona: Universitat Oberta de Catalunya. Recuperado de <http://hdl.handle.net/10609/63466>

Sommerville, I. (2011). *Ingeniería de software* (9ª ed.). México: Pearson Educación de México.

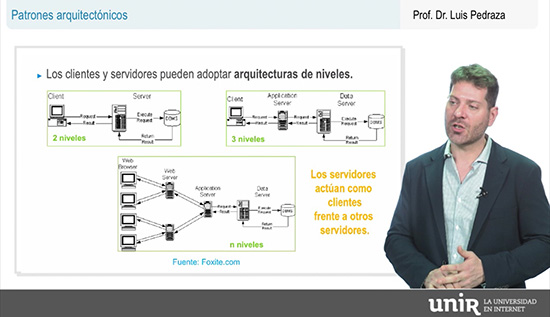
Tanenbaum, A. S. & Steen, M. van. (2007). *Distributed systems: principles and paradigms* (2ª ed.). Upper Saddle RIiver, NJ: Pearson Prentice Hall.

University of California. (2018). SETI@home [Web]. Recuperado de <https://setiathome.berkeley.edu/>

A fondo

**Lección magistral: Patrones arquitectónicos**

En esta lección magistral pretendemos dar una visión general de algunos de los patrones de diseño arquitectónico más habituales, resaltando su utilidad e importancia. Además, comentaremos la influencia que estos patrones tienen sobre los requisitos no funcionales de un sistema, y explicaremos algunas de las arquitecturas más habituales: cliente–servidor, tuberías y filtros, arquitecturas de capas y modelos de repositorio.



Accede a la lección magistral a través del aula virtual.

Arquitectura de software

En este vídeo el Profesor Demián Gutiérrez de la Universidad de Los Andes ofrece una introducción al concepto de arquitectura de software. También explica la importancia de una buena arquitectura de nuestro sistema, en cuanto que afecta a factores como la confiabilidad, rendimiento, escalabilidad o reusabilidad.



Accede al vídeo a través del aula virtual o desde la siguiente dirección web:

<https://youtu.be/ZjxTNGpgP4k>

Arquitectura del software multicapa

En este vídeo el Profesor Antonio Garrido de la Universitat Politècnica de València realiza una introducción al concepto de arquitectura del software o arquitectura del sistema. A continuación, explica de las principales características de las arquitecturas multicapa y, finalmente, muestra un ejemplo de diseño.

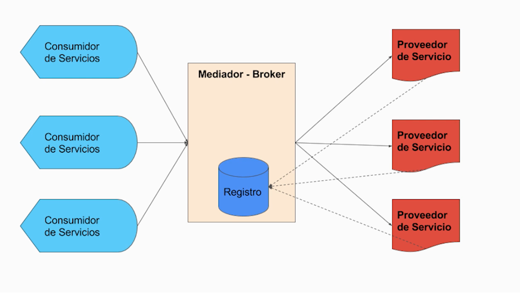


Accede al vídeo a través del aula virtual o desde la siguiente dirección web:

<https://youtu.be/kHvxX1E9vIU>

Arquitectura de Software *Peer To Peer*–Patrón Bróker

En este vídeo Lenin Lozano realiza una introducción a los conceptos que subyacen en las arquitecturas P2P. También hace una introducción al patrón bróker, y su papel e importancia en la comunicación dentro de este tipo de arquitecturas.



Accede al vídeo a través del aula virtual o desde la siguiente dirección web:

<https://youtu.be/j88j77S9Q7U>

Test

1. ¿Por qué los requisitos no funcionales son importantes a la hora de definir la arquitectura de un sistema?

A. Porque la arquitectura de un sistema determina características como el rendimiento general, la protección o la disponibilidad.

B. Porque uno de los requisitos no funcionales debe ser siempre la arquitectura del sistema.

C. Porque imponen restricciones sobre la metodología de desarrollo que debe emplearse.

D. Ninguna de las respuestas anteriores es correcta.

1. ¿Qué entendemos como protección de un sistema?

A. La capacidad del sistema para protegerse de amenazas o ataques externos.

B. La capacidad del sistema para evitar que se produzcan fallos, o recuperarse ante la ocurrencia de un fallo.

C. El nivel de rendimiento general del sistema.

D. La fracción del tiempo total que el sistema está funcionando correctamente y proporcionando a sus usuarios los servicios solicitados.

1. ¿Cuáles de las siguientes es una arquitectura apropiada para favorecer la mantenibilidad del sistema?

A. Una arquitectura fuertemente centralizada, donde todos los datos y capacidad de cómputo se acumulan en una única máquina.

B. Una arquitectura de capas, donde cada una de las capas puede ofrecer servicios a otras.

C. Una arquitectura repartida entre componentes de granularidad pequeña o mediana y débilmente acoplados entre sí.

D. Una arquitectura de N versiones de software.

1. ¿Qué entendemos como un «cliente pesado»?

A. Aquel que se conecta con un servidor empleando protocolos antiguos no estandarizados.

B. Aquel que solo contiene la capa de presentación de datos al usuario y ofrece interactividad con el sistema.

C. Un cliente que solo puede ejecutarse en PC de escritorio, pero no en dispositivos más ligeros como tabletas y móviles.

D. Un cliente que, además de contener la capa de presentación, contiene lógica de negocio y almacena y manipula datos.

1. En un patrón de arquitectura de «tuberías y filtros», ¿qué es un filtro?

A. Cualquier elemento que se interpone entre un cliente y un servidor y analiza los datos enviados para bloquear o no la petición.

B. Un componente que recibe unos datos de entrada, los procesa, y proporciona unos resultados.

C. Un componente que se conecta con otro a través de algún mecanismo de comunicación, de manera que es posible procesar datos de manera encadenada e incremental.

D. Las respuestas B y C son correctas.

1. ¿Cuándo es apropiado emplear arquitecturas de repositorio?

A. Cuando es necesario procesar grandes cantidades de información en lote, realizando varios tipos de procesamiento.

B. Cuando es necesario almacenar grandes cantidades de información durante mucho tiempo, que es consumida, o generada, por un conjunto de clientes.

C. Cuando es necesario establecer diferentes niveles de acceso a un sistema de información.

D. Todas las respuestas anteriores son correctas.

1. ¿Qué es un patrón arquitectónico de sistema de protección?

A. Un modelo arquitectónico en el que un elemento independiente supervisa el correcto funcionamiento del sistema.

B. Un patrón de diseño en el que todas las operaciones críticas se centralizan en una única máquina.

C. Un modelo arquitectónico por capas, que permite mejorar la protección del sistema frente a ataques externos.

D. Las respuestas A y C son correctas.

1. ¿Qué ventajas ofrecen los sistemas distribuidos frente a los centralizados?

A. Es más fácil compartir recursos entre diferentes subsistemas.

B. Su modelado e implementación es más sencillo.

C. Permiten la concurrencia, al poder ejecutar procesos simultáneamente en varias máquinas.

D. Las respuestas A y C son correctas.

1. ¿En cuál de las siguientes condiciones resulta más apropiado emplear una arquitectura de dos niveles cliente–servidor con cliente ligero?

A. Cuando es necesario un procesamiento intensivo en el lado del cliente (como en aplicaciones de visualización de datos).

B. Cuando tenemos aplicaciones con un número enorme de usuarios.

C. Cuando tenemos un sistema heredado donde todas las operaciones de lógica de negocio y gestión de datos están ubicadas en el mismo subsistema y es difícil separarlas.

D. Cuando es posible que el cliente pierda la conexión a la red y deba trabajar con datos almacenados en caché.

1. ¿Qué es un sistema P2P semicentralizado?

A. Aquel que contiene un subconjunto de nodos especiales que, además de sus funciones habituales, sirven como directorio para el resto.

B. Aquel en el que todos los nodos son equivalentes y se comportan del mismo modo.

C. Aquel que contiene un único servidor central al que todos los nodos deben acceder para comunicarse con el resto.

D. Aquel en el que los nodos se distribuyen en una arquitectura multicapa.