Tema 1 - Introducción

TECNOLOGÍAS DE LOS SISTEMAS DE INFORMACIÓN EN LA RED, 2022/2023

Profesores TSR
ETSINF – DSIC - UPV

Objetivos

- Entender por qué todo sistema que utilice una red de intercomunicación es un sistema distribuido.
- Identificar qué es un sistema distribuido, por qué son relevantes y cuáles son sus aplicaciones principales.
- Conocer algunos ejemplos de sistemas distribuidos.
- Estudiar la evolución de los sistemas distribuidos escalables e identificar la computación en la nube ("*cloud computing*") como la etapa actual de esta evolución.

CONTENIDO

1 CONCEPTO DE SISTEMA DISTRIBUIDO

Un sistema distribuido es un conjunto de agentes autónomos, que interactúan para alcanzar algún objetivo común.

- Cada agente es un proceso secuencial, con su propio estado independiente, que avanza a su propio ritmo. Al interactuar con el resto pueden hacerlo mediante intercambio de mensajes o usando memoria compartida.
- El objetivo común de la cooperación puede usarse para evaluar el comportamiento global del "sistema".

En la práctica, un sistema distribuido es un sistema en red.

2 RELEVANCIA

El estudio de las propiedades de los sistemas concurrentes nació de la necesidad de entender cómo coordinar actividades paralelas que se desarrollaban sobre un conjunto de recursos (principalmente memoria) compartidos. Esto fue necesario con el fin de asentar sobre bases sólidas las estrategias de implementación de sistemas operativos que permitían compartir los recursos de un ordenador entre actividades concurrentes.

Los sistemas distribuidos y en red forman un caso de especial de los sistemas concurrentes. La característica principal que diferencia a un sistema distribuido dentro de los sistemas concurrentes es el hecho de que cada uno de los agentes tiene su propio conjunto de recursos y en el caso más común, deben cooperar utilizando alguna "red de comunicaciones" que permita el intercambio de información mediante envíos y recepciones explícitas de mensajes.

En sus inicios, estudios sobre las propiedades de los sistemas distribuidos debían comenzar con algún tipo de justificación acerca de por qué era interesante fijarse en estos sistemas que parecían aportar más problemas que soluciones (es complicado coordinar a los agentes en un sistema distribuido).

La justificación que se daba era cuádruple, y sigue siendo válida: por un lado se decía que aumentaba los escenarios de uso (funcionalidad) de los ordenadores, resultado de la posible cooperación entre sistemas a priori autónomos.

Por otro lado, se argumentaba el potencial de incrementar el grado de aprovechamiento de los recursos, mediante su compartición en una red: los recursos disponibles en un ordenador (p.ej., impresoras, discos...) podrían ser accedidos desde cualquier otro ordenador, aumentando las oportunidades de trabajo de dichos recursos y su rentabilidad.

Adicionalmente, se argumentaba, que era la única forma de aumentar el rendimiento obtenible a la hora de realizar un cálculo, pues hay límites físicos a la potencia que un solo ordenador puede aportar (velocidad, memoria, número de cores…). La idea básica consiste en seleccionar una actividad (problema) compleja, y dividirla en tareas (subproblemas) para poder asignar cada tarea a un ordenador diferente. Dependiendo del problema a resolver, esta división/asignación tenía mayor o menor potencial de aprovechar los recursos de computación disponibles.

Finalmente, se aducía que era el único modo de aumentar la fiabilidad de los sistemas al, potencialmente, protegerse de fallos localizados en un sistema repartiendo la información y cómputo entre sistemas cuyos modos de fallo podían no estar correlacionados, con lo que el fallo de un sistema no implica el fallo de otro, ejecutando una copia del cómputo, que así perviviría.

Esta situación cambió relativamente rápido a lo largo de los 80, cuando se empezó a implantar el modelo cliente/servidor entre elementos simples (e.g., de una estación de trabajo a una impresora), sustentado en la existencia de tecnologías de red local variadas.

En los 90, la aplicabilidad de los resultados sobre sistemas distribuidos se vio aumentada al realizarse la posibilidad de crear agrupaciones de un número limitado de ordenadores para aumentar la fiabilidad del conjunto, obteniendo los llamados *clusters* de alta disponibilidad.

Con el tiempo, la tecnología de red local que ganó ha sido Ethernet (con todas sus variantes), y con esta tecnología, el protocolo de red más adoptado sobre ella: el IP (Internet Protocol).

La aparición y desarrollo de la Web en los años 90 popularizó enormemente la adopción del protocolo IP, y permitió aumentar las posibilidades que los entornos basados en conjuntos autónomos de ordenadores comunicantes permitían.

Hoy en día podemos afirmar que todas esas razones mantienen su vigencia porque el entorno de computación actual ESTÁ distribuido e interconectado, con infinidad de "ordenadores" de tipos y funcionalidades diferentes (incluyendo tablets, teléfonos móviles, smart watches, sensores, …), conectados que interactúan para ofrecer o consumir infinidad de servicios remotos a los que acceder como recursos compartidos, como la Web.

De entre los desafíos que poseen importancia en este ámbito destacamos ilustrativamente dos:

1. Cómo aprovechar la conectividad para obtener resultados útiles. El desarrollo de soluciones con estos recursos supone un replanteamiento en el que las técnicas convencionales pierden validez. El cambio de escala puede ser tal que la interpretación de los problemas y la concepción de sus soluciones no se parecen a ninguna de las referencias que poseemos.
    - Imagina que a partir de la versión tradicional de la criba de Eratóstenes para calcular números primos se desea desarrollar otra en la que un millar de ordenadores colaboren. Los nuevos problemas que aparecen afectan al rediseño del algoritmo (¿cómo "dividir" el trabajo?, ¿cómo "reunir" los resultados?), al balance entre actividad y comunicación (¿un mensaje para cada trabajo o resultado?), a la modificación dinámica del número de intervinientes (un ordenador falla), y si, tras todas estas complicaciones, la mejora producida es significativa.
2. Cómo crear subsistemas capaces de proporcionar servicios robustos. Las nuevas tecnologías permiten abordar problemas y escalas anteriormente impensables, en las que aparecen nuevos retos, algunos de los cuales (no todos, por supuesto) ilustramos a continuación:

- Procesar ingentes cantidades de datos con el fin de encontrar aquellos relevantes en un momento determinado
  - Por ejemplo, ¿Cómo se las apaña Google para implantar su servicio de búsqueda?
- Adaptarse a una gran cantidad de demandas de servicio, posiblemente variable en el tiempo, es decir, cómo crear sistemas escalables.
  - ¿Cómo gestiona Dropbox el uso compartido de ficheros por parte de millones de usuarios?
- Conseguir la cooperación efectiva de agentes gestionados independientemente para aprovechar su potencia de cálculo de forma efectiva
  - ¿Cómo distribuir entre millones de voluntarios la simulación de nuevos fármacos contra el cáncer?
- Soportar los fallos (que inevitablemente van a suceder) de los computadores involucrados en cualquiera de los escenarios anteriores, sin obtener resultados erróneos o dejar de llevar a cabo la función.

3 ÁREAS DE APLICACIÓN

Las áreas de aplicación más destacables de los sistemas distribuidos son:

1. *World Wide Web* y computación en la nube.
2. Redes de sensores
3. *Internet of Things (IoT)*
4. Computación cooperativa
5. *Clusters* altamente disponibles

Vamos a tratarlas a continuación…

3.1 Aplicación a WWW

Basada en el modelo cliente/servidor, el servidor espera peticiones de documentos, mientras los clientes son los navegadores web, que envían y reciben documentos.

- Las peticiones a servidores implican la lectura, modificación, creación o eliminación de un documento.
- Los navegadores analizan el documento de hipertexto buscando metadatos, entre los que destacan los enlaces (*links*) que apuntan a otros documentos, que pueden hallarse en éste u otro servidor

Se trata de un paradigma simple y potente, inicialmente diseñado para compartir documentos, pero extendido para permitir que las peticiones sobre documentos se conviertan en peticiones de servicio, de manera que los “documentos” retornados incluyen el resultado de la petición efectuada.

Esta extensión, potenciada por avances en las tecnologías de virtualización (máquinas virtuales, hipervisores,…) y de ejecución de código en el navegador (la llamada web 2.0), han posibilitado la entrega de servicios de software basados en un nuevo modelo de entrega del mismo asentado

sobre el "sencillo" patrón de la WWW, a los que colectivamente se les identifica como "servicios en la nube" (*cloud computing*), en los cuales nos extenderemos más adelante.

3.2 Aplicación a redes de sensores

Estrictamente, un sensor no tiene por qué ser un elemento de cómputo, ni tener mecanismos de comunicación (entendida como comunicación a través de redes digitales). De hecho, el sensor típico es capaz de utilizar algún efecto físico-quimico con el fin de estimar algún parámetro físico (p.e., temperatura, humedad, frecuencia de vibración, decibelios…) del entorno en el que se sitúa, y transformarlo en otro parámetro físico (p.e., voltaje, intensidad eléctrica…) fácilmente medible y transformable a un formato digital.

El abaratamiento de la electrónica capaz de proporcionar una cantidad aceptable de cómputo, y una conectividad a redes de suficiente capacidad (incluso redes IP), ha hecho posible la creación de lo que puede considerarse como mini-ordenadores de propósito específico ("motes") que incorporan sensores, y que son capaces de digitalizar y transmitir los resultados de sus mediciones.

Muchos pueden encontrarse empotrados en dispositivos de uso cotidiano (p.ej., en algunos electrodomésticos) y pueden contener también actuadores (al menos un sistema de avisos), lo que les capacita para un amplio rango de aplicaciones potenciales como vigilancia, detección de desastres (químicos, biológicos...), monitorización del consumo eléctrico y otras.

3.3 Aplicación a la "*Internet of Things*"

Hoy en día no es posible concebir un sistema informático sin pensar en su conexión a una red de comunicaciones. De hecho, casi se está empezando a asumir que cualquier dispositivo electrónico que genera o maneja información debe estar conectado a internet.

Tanta es la dependencia de cualquier elemento informático de algún sistema de comunicación, que se ha acuñado un nuevo *palabro* para denotar el extremo al que nos estamos encaminando: *El internet de las cosas* (IoT), con la visión de que cualquier dispositivo que maneje información quede conectado a Internet, a través de la cual pueda ofrecer y pedir servicios e información, y actuar sobre su propio entorno físico, o el de otros dispositivos también conectados.

La IoT puede ser estudiada como una generalización de las redes de sensores en la que todos los dispositivos pueden interactuar entre sí y alterar su entorno físico.

El objetivo en el fondo es el mismo: la captación de la máxima cantidad de información posible y relevante para su procesado (muchas veces en "tiempo real") con el fin de automatizar procesos nuevos o existentes, pero que pueden llevarse a cabo con mayor efectividad debido a una mayor cantidad de datos de referencia, y a una mayor capacidad de procesamiento rápido y robusto, no dependiente de errores humanos.

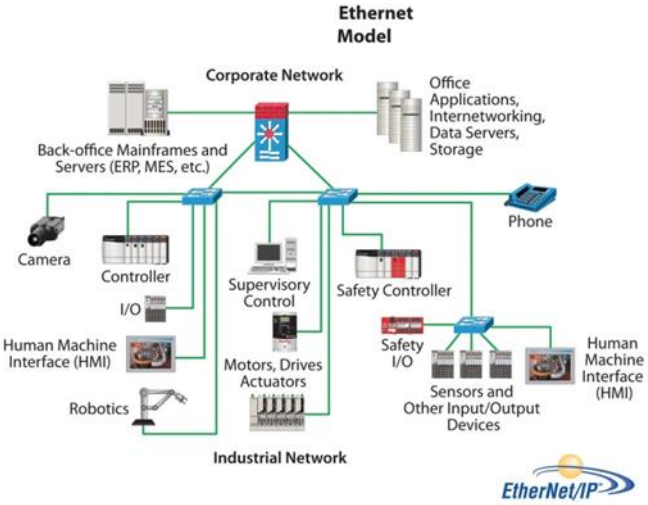
Esta aproximación abre nuevos escenarios:

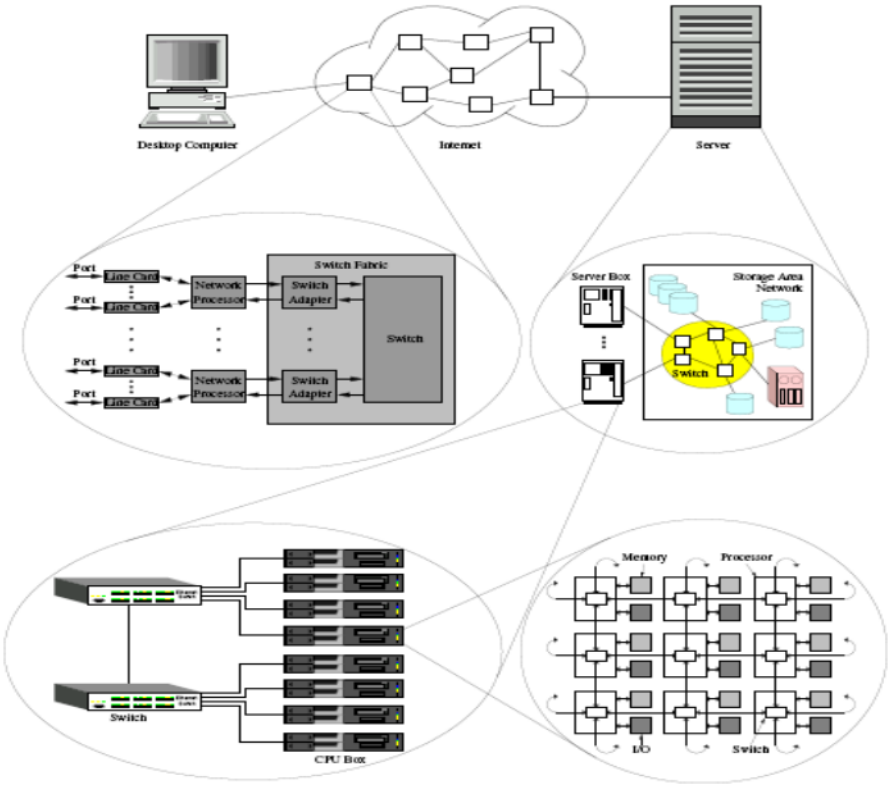
- Ciudades inteligentes
- Automatización de múltiples procesos (construcción, fabricación...)
- Cuidado médico informatizado

- Automatización industrial
- Vigilancia de la salud
- …

En este entorno los sistemas distribuidos intentan facilitar la conectividad e interoperabilidad de todos los dispositivos

El concepto de Internet de las cosas quiere capturar las amplias posibilidades funcionales que se abren cuando se habilita que casi cualquier dispositivo tenga algún modo de cooperar con cualquier otro dispositivo.





3.4 Aplicación a la Computación cooperativa

La mayor parte de los recursos computacionales se infrautilizan, destacando los ordenadores personales, que pasan muchas horas diarias sin hacer nada.

Por otro lado, muchos problemas científicos e ingenieriles pueden dividirse en piezas menores (tareas)

- Cada tarea puede resolverse en un intervalo breve, y los resultados de todas las tareas se pueden combinar para construir la solución completa del problema.
- El servidor crea el conjunto de tareas para el problema, y los clientes pueden obtener una instancia de alguna de esas tareas

Los ordenadores con acceso a Internet pueden suscribirse actuando como "voluntarios" para recibir tareas que resolver

- Instalan un cliente especial: el "runtime" para ejecutar tareas
- El cliente se registra en el servidor, e intercala fases de procesamiento en desconexión con otras de intercambio de datos y resultados con el servidor
- El servidor distribuye tareas entre los clientes registrados y recoge sus resultados

3.5 Aplicación a *clusters* altamente disponibles

El desarrollo del *internet de las cosas* parte del evidente interés de escenarios útiles que se producen cuando se consideran las comunicaciones entre dispositivos "*inteligentes*" (que manejan información), capaces, en algunos casos, de interactuar con el entorno físico.

Los clusters de ordenadores surgen de la necesidad de aumentar la fiabilidad de los sistemas, haciéndolos capaces de soportar fallos sin corromper los datos o parar el procesamiento normal.

Todo dispositivo está sujeto a fallos en su comportamiento. Los ordenadores no son una excepción. Es más, un ordenador está sujeto a fallos de diversos tipos: unos provocados por el hardware mismo, otros provocados por la imposibilidad de asegurar el funcionamiento 100% correcto de todo el software necesario para poner en marcha las aplicaciones.

Desde el principio, uno de los objetivos de las estrategias de manejo de fallos fue la conservación de la consistencia de datos. Varios tipos de sistemas transaccionales se encargaron de asegurar que no se corrompieran los datos que debían persistir cuando aparecían fallos que interrumpían las actividades de actualización. Al menos debería ser posible detectar la corrupción evitando la propagación de información incorrecta.

Con el tiempo se vio la posibilidad de usar varios ordenadores en configuraciones diversas para, además de garantizar la integridad de la información almacenada, permitir que el sistema resultante no parase aunque alguno de sus componentes lo hiciese. Ello era posible por la redundancia de medios puestos a disposición de una computación. Si en lugar de un ordenador se dispone de dos, en principio el fallo de uno de ellos puede ser compensado por el otro, que empezaría a responder a las peticiones antes manejadas por el ordenador caído. Esta descripción aprovecha la propiedad de que no todos los elementos de un sistema fallan a la vez, salvo excepciones derivadas de dependencias mal planteadas (todos dependen de la misma instalación eléctrica, por ejemplo) o inevitables (un terremoto arrasa el edificio donde se encuentra la empresa).

Esta estrategia ha terminado dando lugar a diversas tecnologías implementadas en lo que se conoce como **clusters de alta disponibilidad**, dedicados usualmente al manejo de grandes aplicaciones de gestión empresarial críticas, donde la disponibilidad y la integridad de los datos es esencial.

Los *clusters* de alta disponibilidad intentan dar una respuesta a la necesidad de garantizar la integridad y disponibilidad de la información. Sin embargo las técnicas utilizadas en su implementación, dadas las exigencias de partida, hacen que dichos sistemas no ***escalen*** adecuadamente.

La escalabilidad es la propiedad que tiene un sistema de ser configurado para poder procesar una mayor carga de peticiones.

Los protocolos de replicación y consistencia utilizados por un cluster de alta disponibilidad típicamente imponen una cota superior en el número de ordenadores que lo forman, penalizándolo en rendimiento si simplemente se añaden más ordenadores al cluster (los protocolos de alta disponibilidad consumen más recursos cuantos más ordenadores hay).

Por lo tanto, la única forma que tiene un cluster de alta disponibilidad típico de escalar consiste en aumentar la capacidad de cómputo/almacenamiento/memoria de cada uno de los ordenadores que componen el cluster. Dados los límites físicos existentes en la construcción de ordenadores, ello implica que o bien sea imposible adaptarse a ciertos niveles de carga, o sea

excesivamente caro por tener que utilizar tecnología novedosa que aún no ha sido amortizada en el mercado.

En cualquier caso se requiere un presupuesto extra para poder disponer de sistemas de este tipo, por lo que únicamente se aplican a los entornos que pueden invertir en este alto nivel de disponibilidad: bancario, empresarial, asistencia médica, instalaciones críticas, …

Desde la perspectiva de la eficiencia de la inversión, los centros de cómputo empresariales se sobredimensionan para dar cobertura al *peor caso*, y como consecuencia se infrautilizan estos recursos asignados en exceso. Es posible hacer factor común de muchos de los servicios básicos y recursos de varias empresas para un uso más eficiente: programas y equipos comunes, sueldos de los ingenieros que administren estas aplicaciones y equipos, coste del suministro eléctrico, coste de la infraestructura de red, etc.

Esta tendencia a la racionalización es uno de los motores que conduce a la computación en la nube ("*cloud computing*")…

4 COMPUTACIÓN EN LA NUBE (CC)

La evolución paralela de necesidades en la construcción de sistemas informáticos está confluyendo en lo que hoy en día se conoce como *Cloud Computing*.

Mientras que el IoT busca maximizar el número de escenarios en que se puede extraer valor de la interacción de dispositivos, y los clusters de alta disponibilidad intentan utilizar sistemas distribuidos para incrementar la robustez, ninguna de estas aproximaciones se centra en hacer la creación y explotación de servicios más eficiente. En estas aproximaciones mencionadas tampoco se presta especial atención a la necesidad de escalar "fácilmente".

En lo que sigue daremos una visión de la evolución del concepto de software como servicio, y cómo esto desemboca de forma natural en la visión actual del Cloud Computing. Ésta será una aproximación detallada en la que abordaremos:

1. Programas y servicios
2. Roles en el ciclo de vida de un SaaS
3. Evolución de los servicios software (mainframes, ordenadores personales, centros de cómputo empresariales, SaaS, IaaS, SaaS sobre IaaS, y PaaS)

4.1 CC: Programas y servicios

Podemos establecer que el objetivo general del CC es "*convertir la creación y explotación de los servicios "software" en algo más sencillo y más eficiente*". Al fin y al cabo los programas siempre se han desarrollado para ofrecer algún tipo de servicio con la ayuda de los ordenadores.

La evolución de la industria informática ha ocultado parcialmente este hecho, especialmente porque el fenómeno de los ordenadores personales ha impuesto un modo particular de interacción de los usuarios con sus ordenadores.

4.2 Roles en el ciclo de vida de un SaaS

Con el fin de simplificar, consideramos estos 4 roles:

a) El desarrollador, que implanta los componentes de las aplicaciones
b) El proveedor de servicios, que decide las características del servicio, los componentes que lo constituyen y cómo debe ser configurado y administrado
c) El administrador del sistema, encargado de que cada pieza de *software* y *hardware* esté en su lugar apropiado y adecuadamente configurado.
d) El usuario, que accede al servicio

4.3 Evolución de los servicios software

Este apartado se basa en los anteriores roles para ilustrar la operativa en cada modelo histórico de servicio software

4.3.1 Mainframes

En los orígenes de la computación, los mainframes eran gestionados por personal especializado, encargado de asegurar que todo estaba correctamente, y de cargar el sistema operativo e, incluso, los programas de aplicación.

Siendo el uso de estos sistemas marginal, no aparecían focos de contención provocados por un exceso de demanda.

Un gran porcentaje de usuarios de dichos sistemas debían desarrollar código para poder extraer valor de los mismos. Los papeles de desarrollador y de usuario se confundían, incluso muchos usuarios ejercían de proveedores de servicios tanto con sus propios programas como con los desarrollados por otros.

- Al no diferenciarse roles, encontramos usuarios implicados en demasiados detalles de la gestión de los servicios que ellos mismos debían utilizar

Por otro lado los equipos se usaban eficientemente, dado que estaban compartidos por múltiples usuarios rebajando el coste por uso de cada uno. La adquisición corría a cargo de la institución propietaria del equipo.

4.3.2 Ordenadores personales

Con el paso del tiempo, el descenso en los costes, el aumento de la potencia de cálculo disponible en el hardware, y los avances en las capacidades de los sistemas operativos, se fue estableciendo lo que se ha conocido como el modelo de informática personal. El desarrollo del software pasa a manos de empresas y organizaciones especializadas, que destinan sus desarrollos a un gran número de usuarios. Estos usuarios deben instalar ese software en sus estaciones de trabajo/PCs, con el fin de extraer valor al conjunto del sistema. Podemos identificar este avance con el nacimiento de la industria del software.

En este modelo, se promueve la idea de que el usuario/poseedor del PC tiene el poder de gestionar libremente su sistema, e instalar una gran variedad de software sobre él. El coste de la compra recae en el usuario-propietario, pero ya no compite con otros posibles usuarios. Esta "sobredotación" provoca que los recursos se aprovechen deficientemente (infrautilización del ordenador).

La otra cara de la moneda en esta aproximación es que el usuario-propietario debe encargarse ahora de administrar su sistema. No sólo encargarse de resolver las incidencias de hardware, sino también asegurarse de que el software que instala está en un estado adecuado para poder ser usado.

Estamos ante un modelo de usuario-como-administrador de sistemas, que aún prevalece en buena medida hoy en día, y cuya complejidad es una seria barrera para un buen número de personas.

4.3.3 Centros de cómputo empresariales

Los clusters de alta disponibilidad no se libran de esta servidumbre. Usualmente instalados en los centros de datos de empresas, éstas tienen la necesidad de contar con personal especializado, y de alto coste, que se encargue de la gestión y puesta a punto del hardware y software que constituye los clusters de alta disponiblidad. La empresa usuaria es pues quien administra y provee los servicios basados en dicho cluster, como en el entorno de ordenadores personales.

En ocasiones se agrega el rol de desarrollador de programas internos, dependiendo del volumen y necesidades de la empresa.

También se puede encontrar una variante basada en mantener estos programas en centros de datos externos, principalmente por motivos económicos:

- Evita el coste de adquisición de los equipos
- Reduce y externaliza el coste de administración y mantenimiento de los equipos
- Evita el coste fijo de consumo eléctrico
- La gestión de los costes informáticos resulta más sencilla

4.3.4 Software as a Service (SaaS)

El software siempre se ha "fabricado" con el objetivo de proveer algún servicio. La acepción "moderna" de *Software como Servicio* (SaaS, en inglés), realmente denota un hecho que siempre ha existido.

¿Por qué, pues, parece una novedad hablar de SaaS?

Por el cambio de percepción. En los últimos 30 años, con la implantación del modelo de informática personal, el software se ha visto como algo que había que manipular para poder obtener alguna función de él.

Sin embargo, de un tiempo a esta parte, la evolución tecnológica de las redes, con la mejora en ancho de banda disponible para buena parte de la población, está haciendo cada vez más factible reducir el papel de usuario-administrador a un mínimo. El objetivo es que el usuario no administre su "sistema PC" para acceder a los servicios del software, sino que dichos servicios le sean prestados directamente a través de su conexión a la red, y usando una variedad cada vez mayor de dispositivos.

Otro elemento relacionado es la mejora de la interfaz universal de web (el navegador) que permite ejecutar localmente interacciones complejas. El fenómeno conocido como "Web 2.0"

destaca por presentar interfaces de usuario más atractivas, y por la reducción en la carga en los servidores, mejorando la escalabilidad.

El último ingrediente destacable es el exceso de capacidad en los centros de datos existentes[1], y que permiten comerciar con la capacidad computacional excedente para clientes externos.

Aparece la figura explícita del proveedor de servicios (o proveedor de SaaS), quien debe encargarse de gestionar el despliegue del software sobre tantos servidores como necesite para acoger a su población de usuarios.

Los **problemas a los que debe enfrentarse un proveedor de servicios** son los siguientes:

- Garantizar la calidad del servicio (QoS, *Quality of Service*), según compromiso con los usuarios del mismo. El acuerdo especifica las características cualitativas y cuantitativas que se deben satisfacer.
- Encargarse de seleccionar los componentes software más adecuados para poder montar el servicio, y satisfacer las garantías.
- Reaccionar frente a las variaciones de demanda del servicio por parte de sus usuarios, de modo que para satisfacer las garantías utilice los recursos justos.
- Reaccionar frente a incidentes en el despliegue, de modo que el servicio no se vea interrumpido.
- Mantener el servicio vivo, mientras realiza actualizaciones del software necesarias a lo largo del ciclo de vida del servicio.

Notar que, hasta ahora, todos y cada uno de los puntos anteriores deben ser llevados a cabo de un modo u otro por un usuario de informática personal (quien es su propio proveedor de servicios), lo que cambia bajo el modelo de servicio SaaS, donde el usuario ya deja de tener esas responsabilidades.

¿Qué ventajas comporta este modelo?

Para el usuario son evidentes: éste se ciñe a su papel principal como usuario del servicio, evitando participar en tareas que en principio no son de su interés, ni para las cuales tiene experiencia. Ello permite la participación potencial de un mayor número de usuarios de un servicio, al bajar la barrera de adopción del mismo, así como racionalizar los gastos asociados a esta tarea, mediante la adopción de procedimientos predecibles, y la automatización de tareas.

Adicionalmente, es posible un ajuste razonable de precios del servicio ya que el proveedor puede alcanzar un gran número de usuarios, y, además, se rebajan potencialmente las barreras de entrada a más de un proveedor, aumentando la competencia posible.

Para el proveedor las ventajas se basan en la economía de escala que puede alcanzar de la explotación de una infraestructura común y unos procedimientos comunes para dar servicio a un gran número de usuarios. El modelo de ingresos también se hace más atractivo para el

[1] Origen de la plataforma en la nube de Amazon

proveedor, pasando de un modelo de venta de licencias de software a uno de suscripción con ingresos periódicos comprometidos (similar a una compañía eléctrica, o telefónica).

No queda tan clara la separación de los demás roles, dado que sobre el proveedor recae la responsabilidad del desarrollo de las primeras aplicaciones, y la realización de tareas de administración de equipos y programas.

Hay que hacer notar que el proveedor podrá alcanzar esas economías de escala si es capaz de racionalizar sus propios costes, lo que implica utilizar/pagar en cada momento por los recursos justos (ni más ni menos) que necesita para poder dar su servicio, y usar sus recursos para dar servicio a todos sus usuarios, en lugar de dedicar unos recursos en exclusiva para cada uno de ellos.

Idealmente, en cada momento, si un proveedor necesita 50 ordenadores, debería pagar por el uso de esos 50 ordenadores. Si más tarde debido a cambios en la demanda, sólo necesita 25, debería pagar por esos 25. Adicionalmente, los 50 (ó 25) ordenadores deben ser utilizados para dar varios servicios a su población de usuarios, repartiendo los costes entre ellos.

En este entorno aparece el problema de la adaptación ágil a la demanda de los usuarios, para lo cual se precisa que el servicio esté construido de forma que pueda reconfigurarse rápidamente para conseguir esta adaptación.

A la capacidad que un servicio tiene de adaptar su consumo de recursos según la demanda con el fin de mantener una cierta Calidad de Servicio se le denomina *elasticidad.*

Veamos algunas de las estructuras de servicios que se han establecido dentro de lo que se conoce como Cloud Computing.

4.3.5 Infrastructure as a Service (IaaS)

Un proveedor de un SaaS concreto está limitado en su capacidad de compartir los recursos que usa. Sólo tiene a la población de usuarios de su SaaS.

Mientras que el modelo de negocio del proveedor puede aconsejar que éste se dote de una serie de recursos dedicados para proveer ese SaaS, una aproximación que le permite gestionar mejor sus costes es el acceso a una infraestructura elástica en sí misma (e.g. va reservando los ordenadores y almacenamiento que precisa en cada momento según la demanda de sus propios usuarios).

Para que un proveedor de SaaS pueda adaptarse *elásticamente* a un coste razonable, deben cumplirse al menos estas propiedades:

1. Debe disponer de un proveedor de infraestructura capaz de cobrarle predeciblemente por el uso exacto que haga de los recursos computacionales.
2. El software del SaaS debe ser capaz de adaptarse al tipo de recursos que el proveedor de infraestructura pone a su disposición.
3. Sus procedimientos deben permitirle determinar la cantidad de recursos que el SaaS necesita en cada momento, y automatizar la reconfiguración del despliegue del software que conforma el servicio.

De un tiempo a esta parte, la iniciativa empresarial ha dado respuesta a la necesidad de ser flexibles en la demanda de recursos computacionales y de almacenamiento, produciendo lo que se conoce como *IaaS* (*Infrastructure as a* Service). Un IaaS es un servicio elástico cuyo foco exclusivo es la provisión de recursos computacionales, de comunicación y almacenamiento (los aspectos hardware) a unos precios unitarios por uso.

El modelo SaaS de servicio elástico se filtra así hacia las capas inferiores (la infraestructura), convirtiendo al proveedor SaaS en usuario del IaaS.

El proveedor de IaaS carga así con la responsabilidad de mantener la infraestructura hardware, ofreciendo al proveedor de SaaS la posibilidad de contratar programáticamente un número arbitrario y dinámicamente cambiante de máquinas, comunicaciones y almacenamiento. Esto es posible gracias a la tecnología de virtualización de equipos, mediante la que…

- La asignación de recursos de cómputo (virtuales) es simple y rápida
- La capacidad de los recursos de cómputo se configura fácilmente
- Es muy sencillo instalar una imagen de sistema sobre una máquina virtual

Un proveedor de SaaS puede así usar los servicios de los proveedores IaaS para incrementar/decrementar el número de recursos dedicados a ejecutar el software de su SaaS, según la demanda proveniente de sus usuarios. Esto hace, a priori, factible el ajuste del gasto incurrido por el proveedor SaaS a aquél imprescindible para satisfacer la QoS comprometida con sus usuarios, permitiéndole predecir con gran precisión sus costes en función de la demanda, reduciendo los costes fijos, y ajustando al máximo el coste para sus usuarios.

Notar que, en consonancia con el punto 2 arriba indicado, el software del SaaS debe ser capaz de *escalar* por unidades de máquina, no por el tamaño de las mismas (p.e. velocidad de CPU, cantidad de memoria …). A este tipo de escalado se le conoce como *escalado horizontal*, en contraposición al escalado típico de los grandes servidores consistente en aumentar la potencia de un servidor concreto, incrementando el número de sus CPUs, velocidades, cache, memoria, etc… y al que se le conoce como *escalado vertical*. Hay que tener en cuenta que el escalado vertical, como hemos mencionado más arriba, tiene sus límites en las posibilidades tecnológicas de construcción de ordenadores.

El proveedor IaaS ofrece un servicio simple (mucho más que los SaaS que se ejecutan sobre sus recursos), por lo que puede aspirar a contar con un gran número de usuarios (proveedores SaaS) que rentabilicen la inversión que necesariamente debe hacer tanto en sus centros de datos como en el software para automatizar su gestión.

4.3.6 SaaS sobre IaaS

IaaS introduce un modelo de “pago por uso”, que constituye una característica central de la computación en la nube, y traslada ese mismo modelo a los usuarios de los sistemas SaaS. Por otro lado IaaS facilita la creación de SaaS que se adapten a la carga generada por sus usuarios, manteniendo la propiedad de *elasticidad*.

Para ser rentable, el proveedor SaaS está obligado a un uso eficiente de los recursos en los que la mayoría de los costes son variables. No hay costes directos por reservar cierta capacidad

(compra o compromiso de pago), y lo que se ahorre beneficiará al usuario SaaS, dando aparición a un mercado competitivo de servicios.

Los proveedores IaaS toman los riesgos de la inversión directa (compra de los recursos físicos) bajo el supuesto de que exista una gran población de proveedores SaaS, quienes, a su vez, facilitarán servicios a un alto número de usuarios SaaS provocando una gran demanda de recursos virtualizados.

El proveedor SaaS todavía desempeña varios roles, además del suyo natural como proveedor de servicios *software*:

- Debe gestionar la asignación de recursos *hardware*
- Debe gestionar las imágenes de sistema a instalar, sus actualizaciones y la base de programas a utilizar sobre esos sistemas
- Debe implantar su propia estrategia de gestión de servicios, desarrollando mecanismos de monitorización y actualización

4.3.7 Platform as a Service (PaaS)

El uso de un IaaS, y de software que pueda escalar horizontalmente cubren los puntos 1 y 2 de los requerimientos establecidos anteriormente para conseguir un servicio elástico.

Sin embargo, para cumplir el punto 3, el proveedor de SaaS aún debe enfrentarse a los siguientes problemas:

1. Detectar en cada momento los recursos necesarios para satisfacer un QoS.
2. Modificar la configuración del despliegue de software que constituye el SaaS, sin producir interrupciones del servicio, incumpliendo su objetivo de QoS.

Lo ideal es contar con un *ejecutivo*: aplicación que inspecciona tanto la información estructural de un SaaS como la información de ejecución del servicio, para tomar decisiones de elasticidad de forma automática, eliminando esa responsabilidad del SaaS.

- A este tipo de ejecutivo sobre sistemas elásticos se le conoce como PaaS (*Platform as a Service*).

Cuando un SaaS es integrado para funcionar sobre un PaaS, el SaaS debe cumplir una serie de *requerimientos* adicionales. En particular, el SaaS debe ahora ser especificado con información extra que pueda ser utilizada por el PaaS para tomar decisiones de escalado (metadatos). También será necesario que los componentes del SaaS se adhieran a unas normas de interacción con el PaaS, mediante un API/protocolo adecuado.

La función realizada por un PaaS sobre un IaaS es similar a la que realiza un Sistema Operativo (el ejecutivo) sobre un ordenador. En ambos casos, el sistema operativo (PaaS) decide qué y cuántos recursos es necesario aportar a cada aplicación en ejecución (SaaS), basando esas decisiones en los recursos disponibles, las necesidades de las otras aplicaciones (otros SaaS), y los metadatos disponibles acerca de la aplicación en ejecución.

El PaaS debe abordar los aspectos siguientes:

- Modelos de configuración y de gestión del ciclo de vida (incluyendo las relaciones de dependencia entre componentes). Los servicios distribuidos son proporcionados por aplicaciones. Cada aplicación estará formada por varios programas, donde cada uno facilita cierta funcionalidad. La configuración del servicio requiere que se especifique qué programas lo componen, qué dependencias existen entre ellos y cómo tendrá que desplegarse ese conjunto de componentes. Esa gestión de la configuración y el despliegue de todos estos elementos la realizará el PaaS. Obsérvese que no guarda ninguna relación con la provisión del servicio (función del SaaS) sino con detalles de administración de esos elementos. El PaaS será el responsable de esas gestiones, liberando al SaaS de un buen número de tareas no relacionadas con su rol principal.
    - Esas gestiones deben estar soportadas mediante mecanismos de **composición** (que indiquen qué programas forman la aplicación distribuida que proporciona el servicio y qué dependencias existen entre ellos), **configuración** (para establecer qué recursos necesitara cada componente/programa y en qué grado; por ejemplo, cuánta memoria RAM debe haber en la MV en la que se ejecutará una instancia del componente, qué ancho de banda debería tener la conexión a utilizar, cuántos núcleos de procesamiento...), **despliegue** (herramienta que a partir de un plan de despliegue, realice la secuencia de tareas necesaria para llevarlo a cabo) y **actualización** (cómo modificar todo lo anterior cuando haya cambios en el entorno del servicio, como puedan ser variaciones en la carga o una modificación del programa utilizado en alguna de las componentes).
- Modelo de rendimiento. Para lograr que el servicio ofrecido sea elástico, debe saberse cómo se comportará en función del nivel de carga recibido y el número de instancias desplegadas de cada una de las componentes que dan soporte a ese servicio. Con este objetivo, conviene elaborar un modelo que relacione todos aquellos parámetros que puedan ser relevantes a la hora de calcular el rendimiento ofrecido por el servicio. Los parámetros a considerar pueden depender de la funcionalidad del servicio, por lo que no es sencillo dar un conjunto general que sea aplicable en todos los casos, pero tanto el tiempo de servicio como la tasa de llegada de solicitudes serán dos de esos parámetros.

  Deberá tomarse ese modelo de rendimiento como una base para realizar decisiones de escalado (esto es, incrementar o decrementar el número de instancias en cada componente) y para ello tendrá que complementarlo con:
    - Monitorización automática de parámetros relevantes. Esos parámetros que definen el modelo deben monitorizarse de manera periódica. Así se determina el estado actual del servicio, pero también se puede guardar un histórico de los valores recientes y observar su tendencia actual. Eso permitirá adoptar las decisiones de escalado con cierta antelación, pues tanto el inicio como la parada de una o más instancias son acciones que requieren varios segundos para surtir efecto.
    - Expresión de puntos de elasticidad. Como se verá en futuros temas, cuando se despliegue un servicio sobre un sistema PaaS deberá establecerse un *acuerdo de nivel de servicio* (o SLA, por sus siglas en inglés). En el SLA se especifica qué límites se imponen sobre ciertas características del servicio a proporcionar (disponibilidad, tiempo de respuesta...). El PaaS, a partir del modelo de rendimiento, deberá establecer qué umbrales en cada uno de los parámetros relevantes habrá que considerar para iniciar una acción de escalado, con el objetivo de minimizar el

número de recursos *hardware* utilizados en el despligue y así minimizar también los costes (tanto para el proveedor como para el usuario). Esta gestión recibe el nombre de "expresión de puntos de elasticidad".

- Reconfiguración automatizada en función de la carga. La carga no se mantendrá constante, sino que irá variando a medida que transcurra el tiempo. Gracias a esa monitorización continua de los parámetros relevantes y a la expresión de puntos de elasticidad, las acciones de escalado serán iniciadas y gestionadas de manera automatizada. Con ello, los servicios desplegados en un PaaS serán elásticos, pues **minimizarán el consumo de recursos** y se **adaptarán a la carga dinámicamente**. Esas dos características definen la **elasticidad**.

4.4 Resumen

(A la derecha se muestra la estructura ideal en niveles)

La computación en la nube (CC) se centra en la eficiencia y la facilidad de uso:

- Compartición eficiente de los recursos
  - Consumir solo lo que se necesite
  - Pagar solo por lo que se ha utilizado
- Adaptación sencilla a una cantidad de usuarios variable
- Facilitar formas sencillas para desarrollar y proveer un servicio

Se han identificado tres modelos de servicios en la nube:

1. *Software as a Service* (SaaS)
   - Su objetivo es facilitar aplicaciones como servicio a un gran número de usuarios
2. *Platform as a Service* (PaaS)
   - Aconsejable para automatizar la gestión de recursos para los SaaS y la fácil creación y despliegue de estos servicios
3. *Infrastructure as a Service* (IaaS)
   - Proporciona elasticidad para los sistemas SaaS

Desde la perspectiva de los usuarios, CC es como un regreso a la era de los "*mainframes*".

5 PARADIGMAS DE PROGRAMACIÓN

Un modelo de sistema básico promueve una comunicación asincrónica mediante paso de mensajes: después de ejecutar eventos de envío, los agentes pueden continuar con su ejecución sin bloquearse. Así mismo, los agentes avanzarán sus propias tareas, manejando sus eventos de recepción cuando éstos sucedan (también de manera asincrónica, sin permanecer pendientes de tales recepciones).

Un estilo de programación apropiado para representar los algoritmos que cada agente ejecute es el siguiente:

1. Cada algoritmo está representado por una colección de acciones [A_1,…,A_n], diseñadas para ser ejecutadas atómicamente (sin interrupciones en los efectos que imponen sobre el estado del algoritmo en ejecución).
2. Cada acción A_i, está asociada a una condición C_i (también conocida como *Guarda*). Cuando la condición C_i sea cierta, la acción A_i estará habilitada y podrá ser seleccionada para ejecutarse. En cada instante, cada agente P selecciona una de sus acciones habilitadas para ejecución.
3. Las condiciones pueden depender del estado local.
4. Las condiciones pueden depender de la recepción de un mensaje.
5. Las acciones pueden enviar un mensaje al final de su ejecución.

Hay varios lenguajes de programación que utilizan estos principios, cada uno ofreciendo una sintaxis diferente para expresar variables, condiciones y acciones.

La especificación típica de algoritmos utiliza alguna variación de la sintaxis del lenguaje C para expresar las acciones, las condiciones y las definiciones de estructuras de datos (incluyendo la estructura de los mensajes) en su pseudocódigo.

En programas asíncronos, las estructuras de datos tienen que ser definidas para influir sobre cómo las acciones se ejecutarán una vez habilitadas. Esto es fácilmente aplicable al especificar algoritmos pequeños o con cohesión fuerte. Aun así, en situaciones reales esta aproximación puede resultar tediosa y propensa a errores.

Para resolver este asunto, algunos lenguajes de programación permiten la construcción dinámica de las acciones disponibles, que habrán sido ya total o parcialmente configuradas con los datos relevantes para su gestión una vez estén habilitadas. Así se reduce la necesidad de mantener datos en un contexto global para permitir su ejecución.

Una forma típica de construir las acciones es como *clausuras* en un lenguaje de programación funcional. La clausura conlleva el acceso a aquellas variables (posiblemente locales a la función) que la función deba consultar o modificar durante su ejecución. La guarda especifica como una condición la conclusión de alguna acción asincrónica (recepción de un mensaje, finalización de una llamada al sistema…). Podrá haber variables “libres” que serán especificadas como argumentos de las funciones. En esos casos las guardas deben referirse a las fuentes de datos para esos argumentos, y el “run-time” debe asociarlas cuando llame a la función (y se emplee su clausura) para su ejecución.

Por ejemplo, en un entorno basado en JavaScript como NodeJS, los “*callbacks*” son clausuras de función. Esas funciones pueden especificar parámetros extras que deben ser “llenados” antes de que el *callback* sea ejecutado (una vez habilitado). Esta estrategia evita la utilización de estructuras globales que mantengan los datos extras necesarios para que la acción sea ejecutada.

5.1 Paradigma concurrente con compartición de estado

Los agentes se modelan como ejecutores de acciones atómicas en cada evento. Si pensamos en un servidor, esto significa que el proceso espera la llegada de un mensaje de petición y entonces

ejecuta ésta hasta su finalización antes de devolver el resultado al cliente, sin admitir ningún tipo de interrupción. Esta fue, de hecho, la forma de programar los primeros servidores.

La ventaja de esta aproximación es que la atomicidad está garantizada: mientras una petición está siendo procesada ninguna otra petición puede interferir.

Aun así, hay un problema con esta aproximación simplista. Los servidores a menudo necesitan realizar peticiones sobre otros elementos: otros servidores o el sistema operativo (que también puede verse como otro servidor). La espera del resultado de esas peticiones bloquearía al servidor y, por tanto, lo dejaría incapaz de atender nuevas peticiones de otros clientes. El resultado final es un sistema con muy baja capacidad de servicio.

Una técnica utilizada para evitar esta situación de bloqueo fue que el servidor principal lanzara un nuevo proceso por cada petición recibida. Este proceso secundario era el que realmente servía la petición del cliente.

La planificación de procesos realizada por el sistema operativo permitía que tanto el servidor principal como los procesos secundarios se ejecutaran concurrentemente. El servidor principal, después de delegar la gestión de la petición en el proceso creado, volvía para esperar nuevas peticiones de los clientes.

Así la capacidad de respuesta del servicio mejora ya que el bloqueo de un proceso secundario no bloquea la capacidad de proceso del servidor principal. Éste seguirá atendiendo peticiones nuevas. El estado global entre el servidor principal y todos los subprocesos era compartido utilizando ficheros (un espacio globalmente accesible dentro de una instancia de sistema operativo). El acceso a esos archivos necesitó ser coordinado por los procesos para implementar la atomicidad en las operaciones utilizadas por la lógica del servidor. La coordinación se obtuvo utilizando mecanismos del sistema operativo para acceder a ficheros.

Las dificultades que entrañaban estos mecanismos de compartición de estado condujeron a un estilo de programación en el que la compartición de estado global se minimizó. Gran parte de la información que necesitaban los procesos secundarios se preparaba para su uso exclusivo (incluyendo la petición del cliente). Así la coordinación a través de ficheros compartidos se redujo al mínimo posible y hubo muy pocos problemas derivados del control de concurrencia entre procesos secundarios.

La generación de subprocesos resulta ineficiente tanto en tiempo como en recursos solicitados al sistema operativo. Consecuentemente otras aproximaciones basadas en múltiples hilos de ejecución (donde la concurrencia se da internamente a los procesos) fueron exploradas. El servidor ya no genera nuevos procesos para gestionar la llegada de peticiones. En su lugar, genera hilos de ejecución concurrentes en su propio espacio de direcciones.

Bajo esta aproximación, cada petición es entregada a un hilo diferente. La compartición de estado se consigue trivialmente entre todos los hilos, pues todos ellos comparten el espacio de direcciones del proceso en el que se ejecutan. Esto facilita un estilo de programación en el que el estado compartido global es utilizado para comunicación y coordinación de hilos, requiriendo el uso de mecanismos de control de concurrencia (semáforos, monitores…) para implantar la “atomicidad” implícita exigida por la lógica del servidor.

Al igual que la generación de procesos, los hilos pueden suspenderse independientemente sin bloquear el resto del servidor, que permanece activo. Además, la gestión de hilos es más eficaz en cuanto a consumo de recursos que la gestión de procesos (el sistema operativo tiene menos trabajo) y el acceso al estado compartido es mucho más rápido.

Este ha sido el patrón predominante para construir servidores hasta el momento. De hecho el uso de múltiples hilos de ejecución se mantiene directamente en algunos lenguajes de programación. Lenguajes sin soporte nativo para los hilos también pueden ser utilizados, enlazando con bibliotecas que faciliten la creación y gestión de hilos, así como el acceso a algunos mecanismos de control de concurrencia.

Aun así hay al menos dos desventajas claras en la utilización de hilos:

1. Los hilos, aun siendo más baratos que los procesos, también consumen un buen número de recursos. La creación y la destrucción de hilos son operaciones relativamente complejas y, aunque puedan utilizarse "pools" de hilos para rebajar los costes de la creación y destrucción, deben sintonizarse cuidadosamente para evitar un uso abusivo de recursos. Además, el desarrollo de mecanismos de control de concurrencia también incurre en notables sobrecargas. Si consideramos que la motivación principal del uso de hilos es permitir su bloqueo independiente, lo que realmente se obtiene es que muchos de los hilos generados no estarán haciendo nada la mayor parte del tiempo.

2. La programación concurrente con muchas variables compartidas es propensa a errores. Tales errores pueden conducir a inconsistencias en el estado (la atomicidad será violada) o al bloqueo del servidor (la atomicidad se garantiza, a expensas de comprometer el progreso). A pesar del énfasis que encontramos en muchos planes de estudio sobre programación concurrente basada en sistemas multi-hilo, la situación no parece que pueda mejorar mucho. Esto impone barreras a la velocidad a la que se puede producir el código y a la fiabilidad del producto final.

5.2 Paradigma asincrónico (o dirigido por eventos)

Los entornos de programación asincrónica operan de modo parecido al modelo de programación guarda-acción presentado anteriormente. No son nuevos, existen desde hace tiempo como sistemas dirigidos por eventos, específicamente en el campo de la interacción persona-ordenador (El usuario está constantemente enviando peticiones al programa que maneja la interfaz de usuario. Esas peticiones han sido tradicionalmente manejadas como eventos asincrónicos dentro de un bucle de eventos).

En la práctica, los entornos actuales de programación asincrónicos utilizan un solo hilo en un único proceso. Por tanto, el estado nunca está compartido por varias actividades y no se necesitan mecanismos de control de concurrencia. Se ofrece un modelo similar al de los servidores de la primera generación, implantados mediante un solo proceso.

Los entornos de programación asincrónica deben superar dos dificultades:

1. Evitar el bloqueo del único hilo disponible en el proceso (reproduciendo la desventaja de los servidores con un solo proceso).

2. Que el desarrollo de los programas sea razonablemente sencillo. Esto requiere:
    a. Una estructuración intuitiva del estado que cada acción necesitará manejar cuando deba ejecutarse.
    b. Una forma fácil de expresar las condiciones.

Para evitar la suspensión de las actividades se requiere que todas las peticiones efectuadas a otros servicios (incluyendo el sistema operativo) en un entorno asíncrono puedan ser, a su vez, asincrónicas (es decir, que no impliquen bloqueos). El reto principal se dará en las llamadas a los servicios del sistema operativo. A menudo, algunas de estas llamadas son bloqueantes y tienen que ser convertidas utilizando técnicas multi-hilo por el propio "run-time" (pero no por el software de la aplicación, que sólo observará un hilo).

Una manera común de evitar el bloqueo es la utilización de "callbacks". Al realizar una petición, uno de los argumentos es una función. Cuando la petición termina, esa función se llama con el resultado de la petición como argumento:

```
function handleResults(r) {
        …
}
readFile(f, handleResults);
next_instruction();
```

Básicamente, lo que está haciendo este programa es:

1. Crea una condición: "La llamada a *readFile* termina".
2. Define una acción mediante una definición de función: *handleResults*.
3. Fija esa condición ("La llamada a *readFile* termina") como la guarda de *handleResults*.
4. Continúa ejecutando el resto del programa (en este caso, empezando con next_instruction), sabiendo que *handleResults* será ejecutado MÁS TARDE (se acaba de crear la condición mientras se ejecutaba alguna acción ATÓMICA: no puede ser interrumpida por otra acción, aunque su guarda fuera cierta y la acción fuera habilitada tan pronto como se creara).

Cuando *handleResults* sea invocada, recibirá los datos generados por la petición *readFile*. Así, el estilo de programación basado en callbacks parece resolver nuestras preocupaciones: evita los bloqueos convirtiendo todas las peticiones en asincrónicas, facilita la creación de condiciones dinámicas (la finalización de peticiones), y facilita el manejo del estado que necesitarán las acciones (parámetros del callback).

En realidad, todavía se necesita: (a) ser prudente con las actividades que necesiten mucho tiempo y (b) prestar un cuidado especial en la gestión de estado.

Las operaciones largas acapararán al único hilo, retardando excesivamente la ejecución de cualquier otra acción habilitada. Esto puede generar sistemas poco eficientes. Si un programador no es prudente, el "run-time" no podrá hacer nada para evitar esta situación.

Para evitarlo, el programador debería dividir esa operación prolongada en fragmentos que admitan ejecución atómica. Esto puede ser fácil con un soporte apropiado del entorno de programación.

Por ejemplo, supongamos que queremos computar la función factorial sin acaparar el único hilo disponible. Podríamos utilizar esta aproximación

```
function factorial(n) {
  if (n == 0) return 1;
  executeLater(factorial, n-1)
}

factorial(10000);
```

En este código, hemos supuesto que el "run-time" tiene una operación, `executeLater`, que toma una función y el argumento que deba recibir, actuando como sigue:

1. Crea una condición que será inmediatamente cierta.
2. Asocia la condición con una acción que consiste en ejecutar la función recibida en el primer parámetro, con los argumentos recibidos en el segundo parámetro.
3. Termina la ejecución de la función.

Así, cuando *factorial*(10000) es ejecutado, lo que habría sido una computación larga y continua dentro del único hilo, será convertido en 10000 acciones muy cortas, dando a las demás acciones habilitadas una oportunidad de ejecutarse.

La gestión de estado se da como resultado del paso de argumentos a los callbacks. Muchos "run-times" permiten acceder al estado global del proceso. El acceso a ese estado global no es concurrente. Desafortunadamente, en algunos casos extremos en los que se realicen otras invocaciones, la hipótesis de atomicidad puede romperse y se debe prestar atención para no introducir inconsistencias. Para ello se deben tomar ciertas precauciones al escribir las condiciones que habilitarán a los diferentes callbacks.

Además, cuando se escriban los callbacks será necesario en muchos casos pasar información adicional, aparte de la que el callback reciba en sus parámetros. Diferentes lenguajes de programación habilitan esto de manera diferente. En entornos de programación funcional asincrónica (como por ejemplo, JavaScript) la forma habitual es pasando "clausuras" como callbacks, donde las clausuras encapsulan el acceso a múltiples ámbitos anidados de definición de variables.

6 La Wikipedia como Caso de Estudio de TSR

Para ilustrar de un modo concreto el tipo de problemas que acompañan a los Sistemas de Información en Red, pasamos a acometer el estudio de un caso concreto del que se van desprendiendo sucesivamente oportunidades y necesidades de mejora, que a su vez provocan nuevos problemas que deben ser abordados. La primera sigla de la asignatura, T, representa las tecnologías aplicables para resolver ese tipo de problemas.

Debe entenderse que la variedad de sistemas bajo estudio no puede ser representada únicamente con un caso, pero que, a título introductorio, facilita un contexto en el que tiene sentido identificar las necesidades y proponer soluciones razonables.

El servicio más conocido que pertenece a nuestro ámbito de estudio es la World Wide Web, conocido tanto por su utilidad universal como por haber sido objeto de estudio en la asignatura de Redes. De entre los servicios web existentes, algunos destacan[2] mundialmente por su envergadura, como:

- el buscador de Google[3], con más de un millón de servidores,
- la red social Facebook[4], con más de mil millones de usuarios,
- la web de los ferrocarriles chinos[5], con más de mil millones de clicks diarios,
- las tiendas online Amazon, eBay y Alibaba, cuyas ventas en 2013 totalizaron[6] casi cien mil millones de dólares,
- los servicios de vídeo de Netflix[7] y YouTube
    - En 2015, Netflix contaban con 60 millones de suscriptores[8] que descargaban 45 GB al mes… ¡cada uno! En EEUU y Canadá representaban el 35% del tráfico total. Estas cifras no hacen más que aumentar. Las agregadas aún más, pues múltiples compañías compiten por la atención de los usuarios, y se expanden a nivel global.
    - Las cifras[9][10] de YouTube son más escalofriantes todavía, ¡con más de mil millones de usuarios, y siete mil millones de reproducciones al día (en 2015[11])!.

Quizá podría pensarse que sean excepciones que requieren soluciones a medida, pero el tipo de problemas a los que se enfrentan no dejan de repetirse en otros servicios más modestos ni en otros menos duraderos:

[2] **Aunque las fechas no sean actuales, los órdenes de magnitud siguen siendo representativos**
[3] https://www.google.com/
[4] https://www.facebook.com/
[5] http://www.12306.cn/, según artículo en http://www.chinadaily.com.cn/china/2012-01/09/content_14406526.htm
[6] http://uk.reuters.com/article/2014/09/19/alibaba-ipo-idUKL1N0RK1KW20140919
[7] https://www.netflix.com/
[8] http://expandedramblings.com/index.php/new-updated-netflix-stats/
[9] https://www.youtube.com/yt/press/statistics.html
[10] http://expandedramblings.com/index.php/youtube-statistics/
[11] http://www.forbes.com/sites/edmundingham/2015/04/28/4-billion-vs-7-billion-can-facebook-overtake-youtube-as-no-1-for-video-views-and-advertisers/

- La popularidad de una web modesta, diseñada para una actividad de magnitud moderada, se ve propulsada por aparecer mencionada en algún medio de comunicación de amplio alcance, ocasionando el "*efecto Slashdot*[12]", que produce una "muerte por éxito" del servicio al no poder responder satisfactoriamente a la demanda.
- Un proyecto nacido para cubrir mundialmente un acontecimiento temporal, debe estar preparado para ofrecer un muy elevado nivel de servicio durante el periodo de vida del acontecimiento. Ejemplos destacables son las retransmisiones deportivas asociadas a torneos de tenis (Wimbledon[13], Roland Garrós[14], …), los JJ.OO. de verano[15], o los campeonatos mundiales de fútbol[16].

Deseamos seleccionar un sitio web en el que identificar los problemas y las soluciones que sus diseñadores han ideado; esto requiere el acceso a una buena documentación técnica, lo cual no es habitual en empresas privadas cuya infraestructura se califica como confidencial. Por otro lado, académicamente, es aconsejable la simplificación de detalles para facilitar la comprensión del funcionamiento global.

A la luz de las consideraciones anteriores, y dada su relevancia social que la hizo merecedora del premio de Cooperación Internacional 2015 de la Fundación Princesa de Asturias[17], la **Wikipedia** es una muy buena candidata.

6.1 Wikipedia en la actualidad

El jurado del premio resume los datos relevantes de la Wikipedia en la fecha de concesión del galardón:

"*Wikipedia es una enciclopedia digital de acceso libre creada en 2001 y que está escrita en varios idiomas por voluntarios de todo el mundo, cuyos artículos pueden ser modificados por usuarios registrados. Utiliza la tecnología wiki, que facilita la edición de contenidos y el almacenamiento del historial de cambios de la página. Fundada por el empresario estadounidense Jimmy Wales, con la ayuda del filósofo Larry Sanger, está gestionada desde 2003 a través de la Fundación Wikimedia.*

Wikipedia comenzó el 15 de enero de 2001 como complemento de una enciclopedia escrita y evaluada por expertos llamada Nupedia. […]. Ambos proyectos coexistieron hasta que el éxito de Wikipedia acabó eclipsando a Nupedia, que dejó de funcionar en 2003. El crecimiento de la enciclopedia, que figura entre los diez sitios web más visitados del mundo, ha sido continuo y

[12] https://en.wikipedia.org/wiki/Slashdot_effect
[13] http://www.wimbledon.com/index.html
[14] http://www.rolandgarros.com/en_FR/index.html
[15] http://www.olympic.org/london-2012-summer-olympics
[16] http://es.fifa.com/worldcup/archive/brazil2014/
[17] http://www.fpa.es/es/premios-princesa-de-asturias/premiados/2015-wikipedia.html?especifica=0&idCategoria=0&anio=2015&especifica=0

actualmente tiene más de 35 millones de artículos en 288 idiomas […]. La Wikipedia en inglés es la más amplia, con más de 4.800.000 artículos. Wikipedia cuenta, actualmente, con más de 25 millones de usuarios registrados, de los que más [de] 73.000 editores son activos y tiene unos 500 millones de visitantes únicos al mes.

La Fundación Wikimedia, creada en 2003 y con sede en San Francisco (EE.UU.), es la que dirige y gestiona Wikipedia."

Teniendo en cuenta[18] el grado de repercusión de la Wikipedia (cerca de 18,500 millones de páginas[19] accedidas al mes, 12º en el ranking de Alexa), su reducido número de trabajadores (unos 400), y su financiación procedente de donaciones (cerca de 170 millones de dólares), se entiende que sus responsables han sido muy selectivos a la hora de invertir en el funcionamiento de su servicio web.

6.2 MediaWiki, el motor de la Wikipedia, es un sistema LAMP

El software que constituye el motor de la Wikipedia ha sido desarrollado por ellos mismos y perfeccionado a lo largo del tiempo. Se trata de un software en constante evolución basado en el concepto de Wiki[20]: *"(del hawaiano wiki, 'rápido')[2] es el nombre que recibe un sitio web cuyas páginas pueden ser editadas directamente desde el navegador, donde los usuarios crean, modifican o eliminan contenidos que, generalmente, comparten*". Se trata de un sistema de escritura colaborativo de uso sencillo pero con grandes capacidades.

La Wikipedia ha impulsado esta tecnología, desarrollando software desde Enero de 2001[21], fecha en que crearon UseModWiki[22], escrito en PERL, sobre LINUX, y almacenando la información en archivos en lugar de base de datos.

Este software fue reemplazado por Phase-II al comienzo de 2002, destacando por usar un motor wiki en PHP, pero en Julio fue de nuevo mejorado (Phase-III = ¡MediaWiki!) para adaptarse al crecimiento del proyecto. Es significativo que hasta Junio del 2003 no se añadiera un segundo servidor, incluyendo en él la base de datos (se separa del servidor web) y las webs en otros idiomas diferentes del inglés. Más detalles en los apartados 1 a 4 del capítulo 12 (MediaWiki[23]) del libro "*The Architecture of Open Source Applications, Volume II*"

Todo este software ha continuado evolucionando, pero comparten un núcleo tecnológico: son sistemas LAMP. Esta combinación gira en torno a sus 4 iniciales: Linux + APACHE + MySQL + PHP.

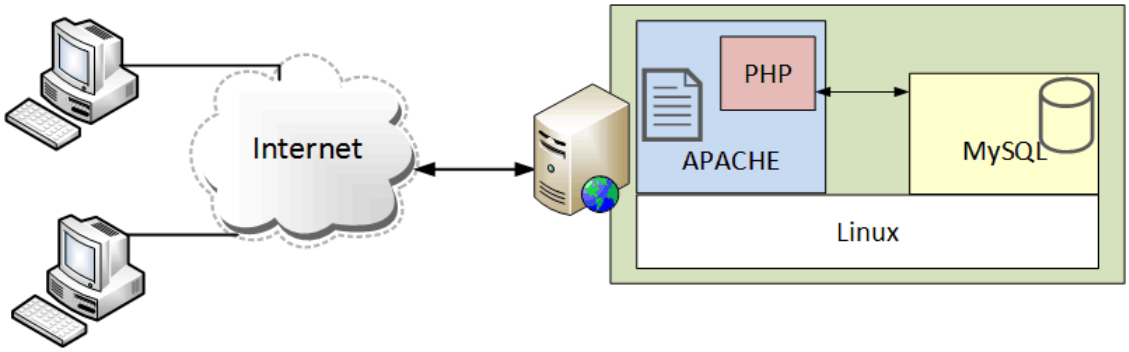
- Linux es un sistema operativo tipo UNIX

[18] https://stats.wikimedia.org/, https://wikimediafoundation.org/about/annualreport/2019-annual-report/, datos de 2019
[19] *pageviews*
[20] https://es.wikipedia.org/wiki/Wiki
[21] https://en.wikipedia.org/wiki/History_of_Wikipedia#Hardware_and_software
[22] https://en.wikipedia.org/wiki/UseModWiki
[23] http://www.aosabook.org/en/mediawiki.html

- APACHE es un servidor de web. Recibirá peticiones HTTP de los clientes, y ejecutará las acciones necesarias para devolver las respuestas
- MySQL es un gestor de bases de datos relacionales
- PHP es un lenguaje de scripting ligero y sencillo, con bibliotecas de todo tipo, y con facilidad para producir documentos de hipertexto y para interactuar con SGBD relacionales.

Una plataforma LAMP encaja con el modelo de 3 capas de las aplicaciones cliente/servidor:

1. La capa de **interfaz de usuario** contiene las componentes que forman las interfaces de usuario de las aplicaciones (lógica de presentación). Se corresponde con los documentos de hipertexto que devuelve APACHE y visualiza el navegador del cliente.
2. La capa de **aplicación** (o lógica de negocio) contiene las reglas de funcionamiento de la aplicación, sin integrar datos concretos. Se corresponde con parte del código escrito en PHP.
3. La capa de **datos** (o de persistencia) contiene la información que los clientes manipulan mediante los demás componentes de la aplicación. Se corresponde con la BD gestionada por MySQL.

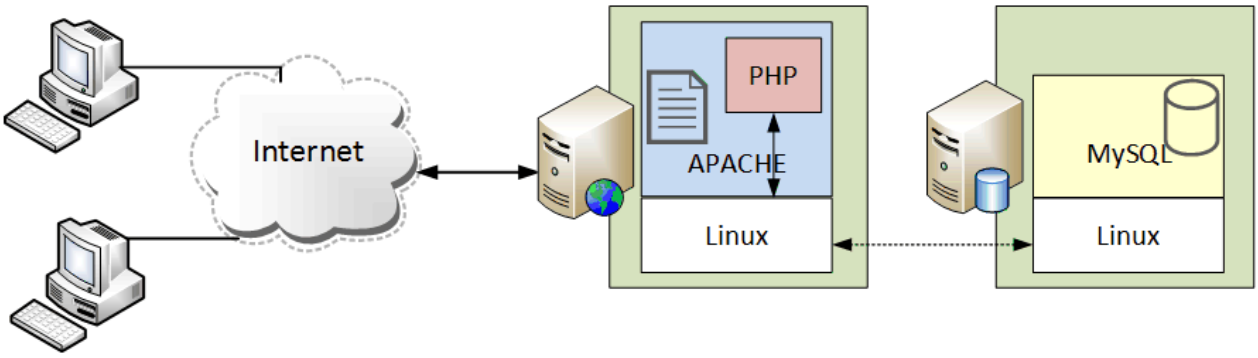


La secuencia **simplificada** de la atención de una petición de un cliente podría ser:

0. Originalmente hay un proceso APACHE y otro MySQL a la espera de recibir trabajo.
1. El cliente envía la petición HTTP al servidor. Dicho cliente queda en espera.
2. El servidor APACHE es despertado por el S.O. para atender la solicitud
3. APACHE determina que debe ejecutarse un programa PHP, al que pasa los datos. Como el intérprete de PHP es una extensión suya (forma parte del proceso), APACHE no queda en espera.
4. El programa se ejecuta emitiendo órdenes SQL que el SGBD recibe. El programa PHP (y APACHE) queda en espera de que el proceso SGBD le devuelva una respuesta.
5. El servidor MySQL es despertado por el S.O., recoge la solicitud y la ejecuta, devolviendo la respuesta al programa PHP. MySQL queda a la espera de una nueva solicitud.
6. Las interacciones 4 y 5 pueden repetirse. Al final, el programa PHP confecciona una página de respuesta que APACHE envía por la red. Finalizada la acción, el programa termina y APACHE queda en espera de una nueva solicitud.
7. El cliente recibe el resultado, y su navegador puede visualizarlo en pantalla.

Por la forma en que se intercomunican, como procesos, no es necesario que el SGBD (actuando como servidor) y el programa PHP (actuando como cliente) se ejecuten en el mismo equipo.

- Esta distinción entre nodo (equipo) y componente (en este caso los servidores APACHE y MySQL) tiene extrema importancia en el planteamiento de un sistema distribuido.
- Sin embargo, salvo cambios sustanciales, no es posible que el programa PHP se ejecute en un equipo diferente del que ocupe APACHE.



Volviendo al tema: ¿qué es MediaWiki exactamente? MediaWiki es un caso concreto de sistema LAMP que incluye scripts para la parte PHP de la ilustración anterior, junto con la configuración necesaria de los servidores, alguna BD y tablas, y algunos contenidos de hipertexto.

6.3 Usando MediaWiki en el contexto de la Wikipedia

La mayor parte de la información estadística sobre la Wikipedia gira en torno al número de editores, entradas e idiomas disponibles. Adicionalmente se pueden encontrar cifras sobre el número de accesos de usuarios lectores, pero deberemos procesar dicha información para averiguar qué recursos se necesitan.

Por ejemplo, una aproximación para conocer los recursos de red necesarios puede consistir, dada una tasa de accesos mensual, en multiplicar el tamaño de página por esa tasa, obteniendo el ancho de banda bruto mensual consumido en lecturas (los datos[24] que se muestran son válidos en 2021).

- Ejemplo de artículo corto en inglés: *albufera* (1,130 KB según la consola de inspección de Firefox)
- Idem de artículo largo[25]: *United States* (7,210 KB)

[24] Datos entre julio de 2020 y junio de 2021 consultados en https://stats.wikimedia.org/#/all-projects/reading/total-page-views/normal|bar|1-year|~total|monthly dan un promedio de 24,000M de accesos al mes

[25] Realmente no es el mayor tamaño, pero en este documento es irrelevante

El consumo de red mensual, se encuentra entre 27,120 TB[26] y 173,040 TB, para los que, tomando un patrón de acceso constante pero irreal, se requieren velocidades entre 83.70[27] Gbps y 534.07 Gbps. Si observamos los momentos de mayor uso según sus estadísticas[28], se alcanzan picos que triplican ese promedio, momento en el que se requiere un **ancho de banda entre 251 Gbps y 1,602 Gbps**. La infraestructura de red no alcanza estas cifras con facilidad.

¿Puede atenderse toda esta demanda desde un único servidor? Pongámoslo a prueba.

6.3.1 Acceso a Internet

Un proveedor de acceso a Internet puede alojar un servidor en sus instalaciones para que aproveche la infraestructura de red de dicho ISP. Tomando datos[29] de EEUU, la mayor velocidad *convencional* la ostentaba la infraestructura de fibra de Google, con 2,000 Mbps.

¡¡¡En el peor caso nos harían falta **800 líneas de la mejor conexión de Internet** del mundo!!!.

Hay que insistir en que los responsables de la Wikipedia no tiran el dinero por la ventana, de modo que debe haber alguna solución razonable.

La primera alternativa consiste en evitar enviar toda esa cantidad de información. Muchas técnicas comunes son aplicables, como hacer que el navegador almacene en su caché elementos que cambian muy raramente, como hojas de estilo, iconos y scripts. Cuando el navegador los encuentra en su caché, puede reutilizarlos sin volver a descargarlos. Otras técnicas aplicables logran reducir aún más la información, pero no conseguimos un ahorro suficiente; al fin y al cabo contamos con millones de clientes y la cache del navegador es propia de cada uno.

La segunda alternativa consiste en "desconcentrar" el servicio. Esto supone dispersar múltiples servidores por las redes mundiales y contar con el ancho de banda agregado de todos ellos.

La caja de Pandora se abre con las palabras mágicas "múltiples servidores": ¿no se trata del mismo servicio?. Esta solución introduce sus propios problemas, destacando:

- ¿los servidores son clones idénticos?, ¿y por dónde "*se entra*"?
- ¿cómo se gestionan las operaciones de escritura de artículos en la Wikipedia?,
- ¿qué ocurre si alguno se desincroniza respecto al resto?, ¿deben comunicarse entre sí o hay una copia central?
- ¿algún servidor de repuesto para casos de fallo?

Los nuevos problemas pueden asumirse si los beneficios exceden los costes, y en el caso de la Wikipedia así es, pues la alternativa, dadas las cifras de tráfico, es no ofrecer el servicio, o

[26] 24,000M de accesos, multiplicado por los dos tamaños de página mencionados (1,130KBs y 7,210KBs)
[27] 24,000M de accesos/mes = 9,259.259 accesos/s, que multiplicamos por 1,130 KB y por 8 bits, y representamos en Gbps
[28] https://grafana.wikimedia.org/?orgId=1
[29] https://www.highspeedinternet.com/resources/fastest-internet-providers, julio de 2021. En septiembre del 2022 el operador Xfinity anuncia 3,000 Mbps

restringirlo severamente. Sin embargo, los problemas presentados no son triviales, y hay que aplicar mucho ingenio para identificar cada uno de los nuevos retos y darles solución.

Podemos afirmar que multiplicar elementos siempre es problemático, porque la fiabilidad de un sistema depende de la de sus componentes y de la estructura con la que se engarzan: a priori, un incremento en el número de piezas constituye un problema potencial. Supongamos que se precisa disponer de 10 réplicas para mejorar el rendimiento, y que la probabilidad de que un día una réplica concreta falle es del 1 por mil (0.1%); ¿cuál es la probabilidad de que no falle ninguna a lo largo de un mes?

- Para que no falle una en un día: 1-0.001
- Para que no falle en 2 días[30]: (1-0.001)*(1-0.001)= $(1-0.001)^{\mathbf{2}}$
- Para que no falle una en un mes: $(1-0.001)^{\mathbf{30}}$=97.04%
- Para que no falle ninguna de las 10 en un mes: $((1-0.001)^{30})^{\mathbf{10}}$=74.07%

A modo de primera conclusión provisional: *cuantos más componentes, menos fiabilidad*; incrementar el rendimiento añadiendo componentes sin más puede (y seguramente va a) empeorar la fiabilidad del sistema en su conjunto.

¿Y cómo se puede arreglar? Con un diseño cuidadoso en el que un componente con fallos pueda ser detectado y reemplazado, haciéndose cargo otro de su carga de trabajo. ¡Es más fácil decirlo que hacerlo!

Como segunda conclusión, *los problemas introducidos por la replicación se solucionan con más replicación*. Una primera "replicación" se dirige a incrementar el rendimiento, aumentando el número de "trabajadores" disponibles para realizar una cierta tarea, mientras que la segunda replicación intenta resolver los problemas de fiabilidad potencialmente introducidos o agravados por la primera.

Estas consideraciones son aplicables, con sus particularidades, a todos los componentes del sistema LAMP en que se basa la Wikipedia.

6.3.2 El servidor web APACHE (objetos estáticos) y los scripts en PHP (resultados dinámicos)

Este software interviene para recibir las solicitudes de los clientes, por el protocolo HTTP, y decidir si debe devolver un contenido almacenado o ejecutar un programa PHP. Una visión simplificada de esta forma facilita la separación de objetivos. En el primer caso el servidor es un intermediario entre el cliente y los contenidos; en el segundo sirve como transmisor de informaciones del cliente (p.ej. una interrogación por el buscador local) a una aplicación que debe construir una respuesta a medida.

La primera parte puede ser acelerada con cierta facilidad: ¡más memoria!. Si se pudiera colocar toda esa información estática en memoria física, podría accederse a la misma con mucha rapidez, aunque a un precio… Como no es posible representar toda la información en memoria, se suele usar algún sistema inteligente que, adaptativamente, acelera el acceso a las informaciones más demandadas. Es una memoria cache llamada *proxy inverso*, que "atrapa" y

[30] que no falle el primero **NI** el segundo

sirve las solicitudes de objetos estáticos (páginas, ilustraciones…). Aceleradores capaces de realizar esta función para peticiones HTTP pueden ser *Squid*, *Varnish* y *nginx*.

¿Pueden colocarse varias instancias si una sola no alcanza el rendimiento necesario?. Ya que se trata de operaciones de lectura independientes entre sí, puede establecerse algún criterio en el que, del nombre del recurso se obtenga directamente el del acelerador que lo contiene.

- Por ejemplo, colocar tantos aceleradores como letras tiene el alfabeto, y que cada uno contenga copia de los objetos estáticos que comienzan por dicha letra.

Ya vimos que la adición de componentes incrementa la probabilidad de error, por lo que deberán asignarse recursos extra para tratar los casos en los que algún proxy inverso pueda fallar.

Tras este filtrado de solicitudes que referencian a objetos estáticos, las que lleguen a los servidores APACHE supondrán la invocación de programas PHP que construyan la respuesta a la petición. Podemos destacar 4 categorías principales:

1. **Escrituras** en las que se crea o modifica un artículo de la Wikipedia: ¿qué pasa con todas las copias de este artículo que pueden existir en los aceleradores?
    - Se debe asegurar la consistencia de las diferentes copias en los diferentes lugares. Habitualmente se enviará alguna señal para invalidar las copias de los artículos y obligar a *refrescar* ese contenido.

2. Páginas que **dependen del usuario** solicitante, que determinan qué contenidos mostrar y qué operaciones son posibles.
    - Todo aquello que se asimila al concepto de sesión HTTP necesita una continuidad, de manera que el mismo servidor que comenzó a atender a un usuario debe hacerse cargo de sus futuras peticiones.
    - P.ej., un usuario puede ir acumulando páginas para construir un libro, o un usuario se ha identificado con un rol específico cuyas capacidades deben ser recordadas.

3. **Consultas para localizar información** dependiendo de algunos términos de búsqueda. Es una actividad propia del gestor de bases de datos, pero puede diseñarse un sistema interno de cache de respuestas, aprovechando resultados que pudieran haberse calculado antes.
    - Hace falta un diseño muy cuidadoso para conocer cuándo un resultado mantiene su vigencia, pero puede ofrecer grandes beneficios para ciertos casos.

4. **Consultas para obtener contenidos asociados**, de forma que, dado un artículo, pueda consultarse información derivada: enlaces que apuntan aquí, información de la página, historial de cambios (con operaciones sobre los mismos), etc.
    - La parte estática es dominante, lo cual facilita una política de *caching*.

Vemos, de nuevo, que un ingrediente fundamental para la aceleración del sistema se basa en mantener copias de informaciones para no volver a acceder a las mismas o a calcularlas. Potencialmente, ésta es una fuente muy importante de inconsistencias.

Esta estrategia de caching reduce las ocasiones en que es necesario ejecutar un programa. La importancia no es menor, pero todavía quedan peticiones que no pueden satisfacerse de esta forma. Aunque su porcentaje no sea alto (p.ej. 12% en los JJOO de invierno de Nagano, 1998), el coste de procesamiento sí que lo es, superando en muchos casos al 50% del tiempo total acumulado.

Aunque existen técnicas específicas, y muy efectivas, de aceleración de programas en PHP, el grueso del tiempo que consumen está determinado por:

- El sobrecoste de atender una petición dinámica. Ya somos selectivos para reducir su frecuencia, pero deben atenderse los casos inevitables restantes.
- El coste de interactuar con el gestor de la base de datos. Este aspecto se trata en el siguiente subapartado.

6.3.3 El servidor de base de datos MySQL

La interacción con la BD en la Wikipedia es vital, no sólo para organizar los artículos que mantiene, sino para recopilar informaciones, gobernar el ciclo de vida de los artículos, representar usuarios, mantener estadísticas, etc.

Hemos visto que una de las primeras medidas adoptadas en la Wikipedia consistió en separar la BD para ubicarla en otro servidor. Obviamente esta acción obligó a plantear una forma de comunicación entre el tándem APACHE-PHP y el servidor MySQL, típicamente mediante IP y puerto.

Las primeras replicaciones del servidor de web conducen a un desequilibrio si se mantiene inamovible el número de instancias, una, del SGBD: acabará exhausto ante la avalancha de peticiones.

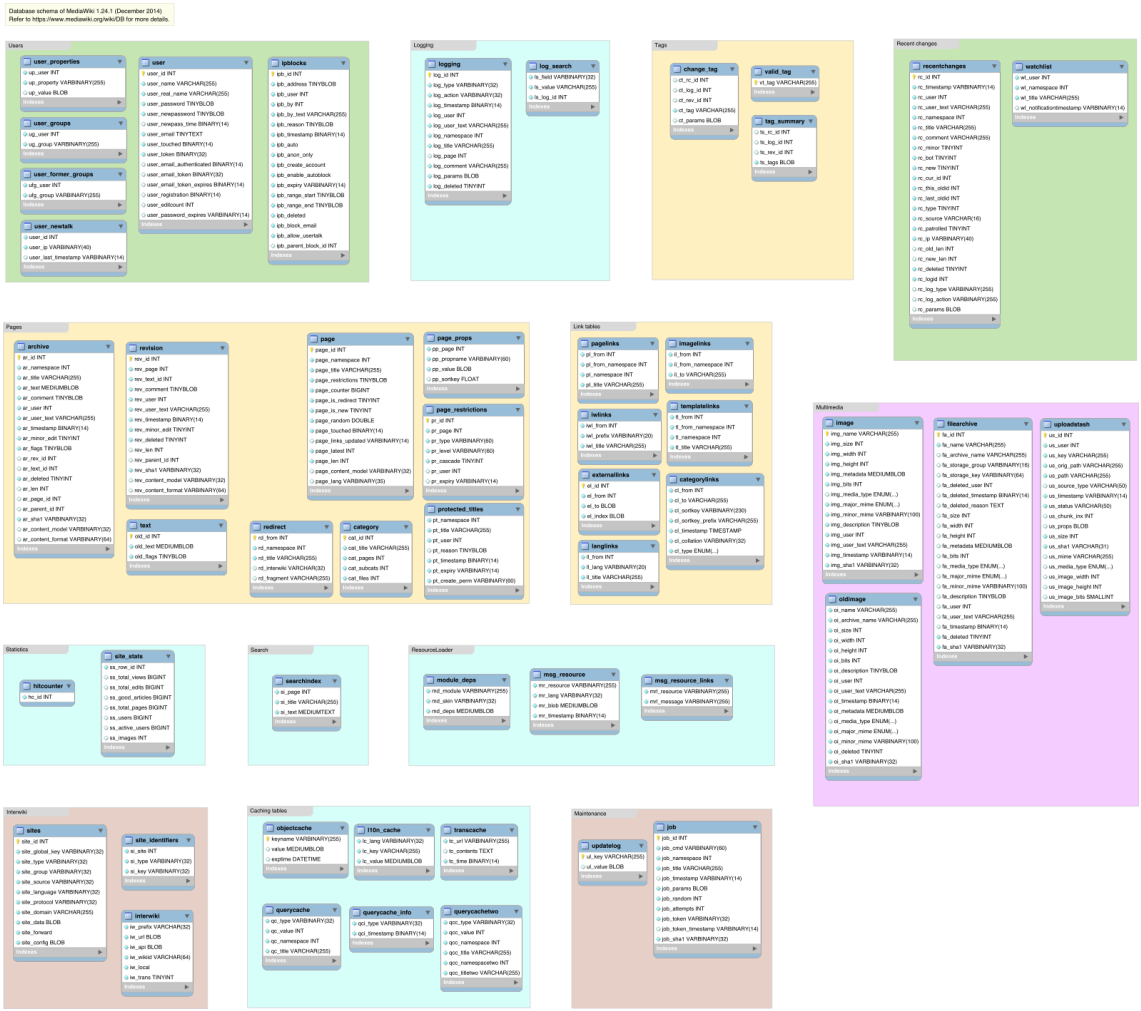
- Imagínese un restaurante con un camarero y un cocinero. El camarero recibe las peticiones de los clientes y se las transmite al cocinero. Si para admitir muchos más clientes únicamente se contratan camareros, puede adivinarse con facilidad dónde se encontrará el *cuello de botella*.

Colocar más instancias del SGBD no es una tarea sencilla si han de mantener la misma información: deberemos descubrir primero si los contenidos pueden separarse en múltiples BBDD independientes.

- Dependerá del uso que se haga de las informaciones y de las relaciones entre sus tablas.

Aunque esta primera parte se haya realizado, seguramente nos quedará todavía un resultado difícil de digerir[31]: 13 BBDD acumulando 50 tablas en Diciembre de 2014.

[31] https://upload.wikimedia.org/wikipedia/commons/f/f7/MediaWiki_1.24.1_database_schema.svg



Las limitaciones no son preocupantes en operaciones de lectura si se compara con las actualizaciones y escrituras: las BBDD que requieren mayor ayuda son probablemente las que sufren más modificaciones, que pueden agruparse en varios niveles:

1. Modificaciones **muy frecuentes**, que suelen acompañar a los accesos de consulta: *Logging* (que añade información en modo *append* sobre los accesos), *Statistics* (su propio nombre lo indica), *Caching tables* (de consumo interno, que acelera el acceso a algunas informaciones de manera adaptativa)

2. Modificaciones **por edición**, son menos frecuentes que las anteriores porque suponen la acción de un editor (¿humano?) alterando el contenido de la Wikipedia: *Multimedia* (representa el depósito de objetos no textuales), *Recent changes* (que mantiene una lista de modificaciones), *Pages* (es la BD de artículos) y *Link tables* (si el nuevo contenido referencia a otros elementos)

3. Modificaciones **eventuales**, que son comparativamente infrecuentes, como dar de alta un usuario (*Users*), y el resto.

La frecuencia de escrituras puede ser importante pero no siempre determinante; p.ej. las escrituras que no condicionan lecturas posteriores presentan escaso conflicto (*Logging* y

Statistics), pero las escrituras cuyos resultados alguien va a necesitar con inmediatez son muy preocupantes (*Caching tables*, *Pages*, …)

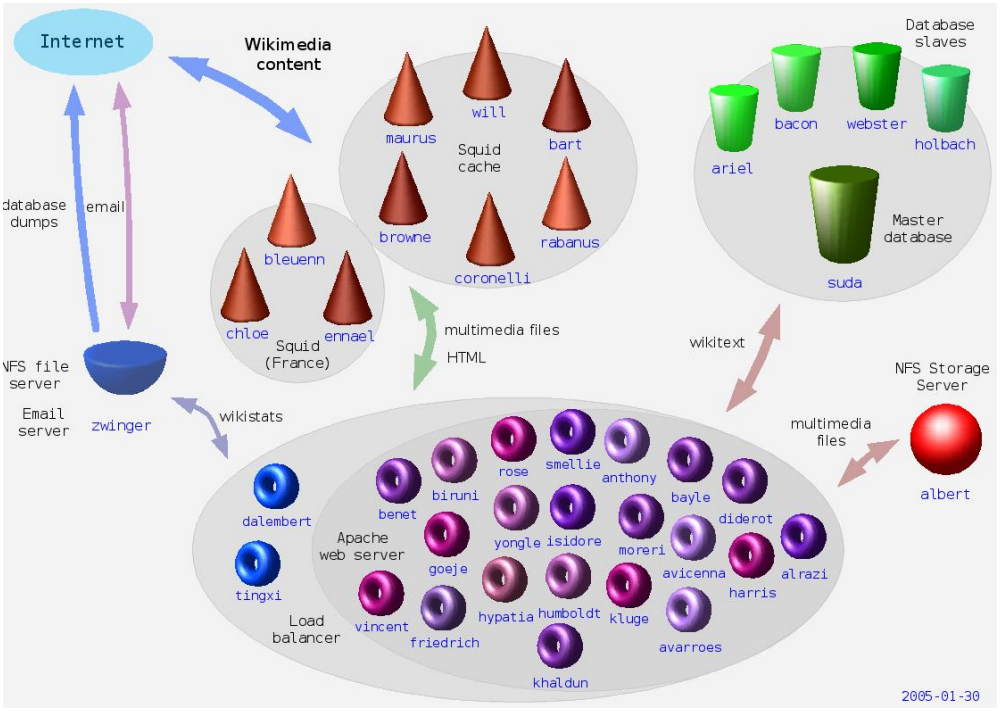
6.4 Arquitectura de la Wikipedia

(La información que se muestra no se encuentra completamente actualizada porque las fuentes[32] tampoco lo están. Pueden consultarse los archivos de configuración en http://noc.wikimedia.org/conf/)

6.4.1 Evolución global de su estructura

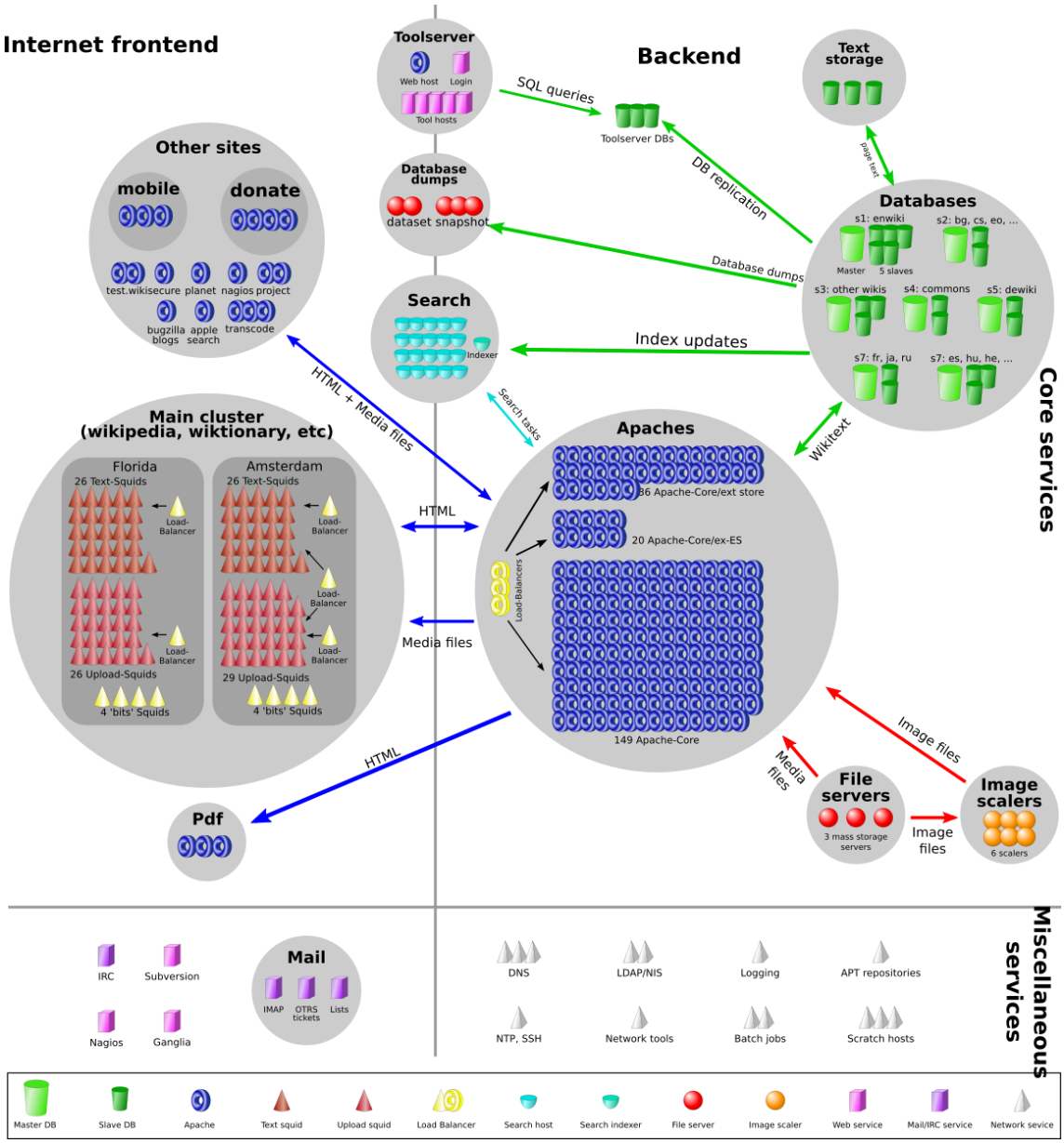
Comenzando con una infraestructura mínima, con un único equipo en el que se aloja y sirve todo el material, la Wikipedia evoluciona en equipamiento, organización y adición de servicios. A modo de ilustración se muestra un par de instantáneas tomadas en 2005 y 2010

- **2005**. Destacable: múltiples servidores APACHE, todavía con nombre propio, dos servicios de proxy inverso, distribución de la BD



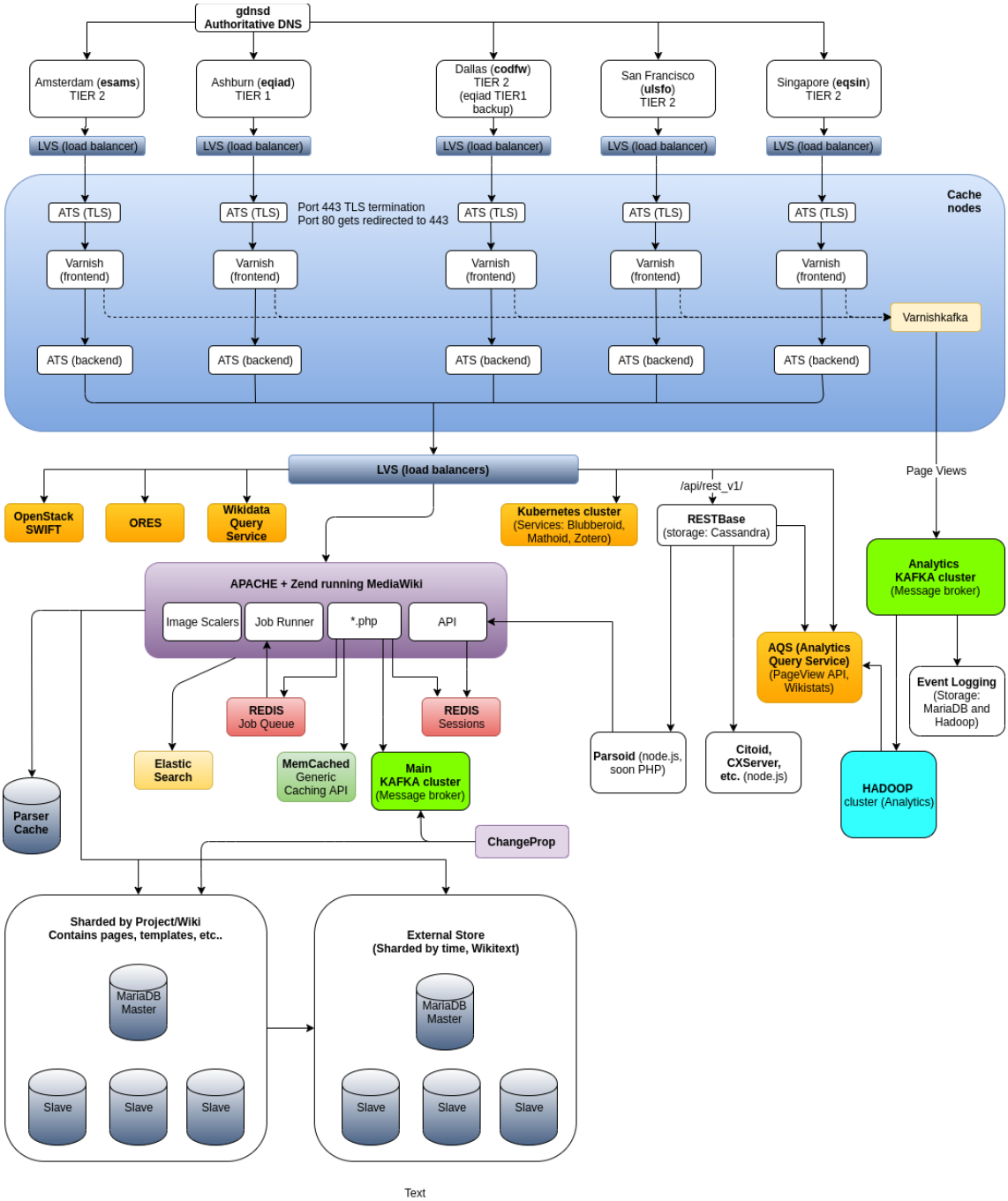
- **2010**. Destacable: hay tantos servidores APACHE que sus nombres pasan a ser números, especialización de sistemas semiautónomos (servicios de proxy inverso, BBDD e indexación), crecimiento de otros sistemas auxiliares, empiezan a cobrar relevancia otros proyectos de la misma fundación.

[32] https://meta.wikimedia.org/wiki/Wikimedia_servers


Internet frontend
Toolserver
Web host
Login
Tool hosts
SQL queries
Backend
Toolserver DBs
Text storage
page text
DB replication
Other sites
mobile
donate
test.wikisecure
planet
nagios
project
bugzilla
blogs
apple
search
transcode
Database dumps
dataset snapshot
Database dumps
Databases
s1: enwiki
Master
5 slaves
s2: bg, cs, eo, ...
s3: other wikis
s4: commons
s5: dewiki
s7: fr, ja, ru
s7: es, hu, he, ...
Search
Indexer
Index updates
Search tasks
Wikitext
HTML + Media files
Main cluster
(wikipedia, wiktionary, etc)
Florida
26 Text-Squids
Load-Balancer
26 Upload-Squids
4 'bits' Squids
Amsterdam
26 Text-Squids
29 Upload-Squids
4 'bits' Squids
HTML
Media files
Apaches
Load Balancers
86 Apache-Core/ext store
20 Apache-Core/ex-ES
149 Apache-Core
Core services
HTML
Pdf
Media files
Image files
File servers
3 mass storage servers
Image files
Image scalers
6 scalers
IRC
Subversion
Nagios
Ganglia
Mail
IMAP
OTRS tickets
Lists
DNS
LDAP/NIS
Logging
APT repositories
NTP, SSH
Network tools
Batch jobs
Scratch hosts
Miscellaneous services
Master DB
Slave DB
Apache
Text squid
Upload squid
Load Balancer
Search host
Search indexer
File server
Image scaler
Web service
Mail/IRC service
Network sevice

6.4.2 Actualidad (datos entre 2019 y 2021)

En el esquema que sirve de referencia, la parte superior (hasta LVS) representa los recursos dedicados a facilitar el acceso mundial a la Wikipedia, mientras que la inferior representa los recursos dedicados al almacenamiento de datos.



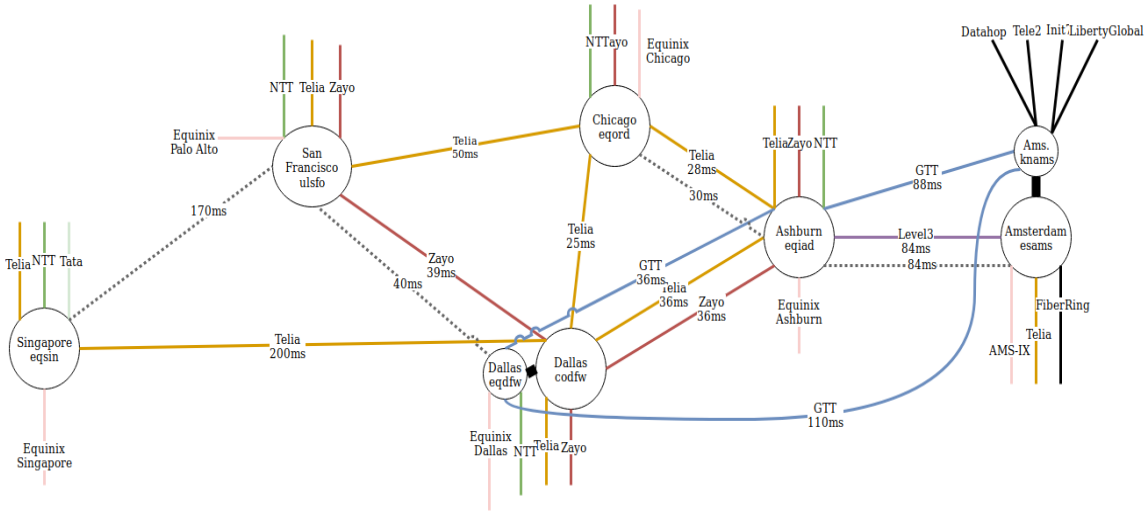
Aunque a continuación se mencionan los componentes más relevantes, un estudio detallado queda fuera de nuestros objetivos.

- **Presencia en Internet**[33]

En Mayo de 2018, los recursos de la Wikipedia se distribuyen entre las siguientes ubicaciones:

- **eqiad** (Equinix en Ashburn, Virginia). Es la instalación principal, e incluye la parte interna de la web y las BBDD.
- **esams** (EvoSwitch en Amsterdam, Holanda), actuando como cluster de cache Squid y otras cosas menores.
- **eqsin** (Equinix en Singapur), actúa como cache para Asia y la zona del Pacífico.
- **ulsfo** (United Layer en San Francisco), actuando como cache para el oeste de EEUU y lejano Oriente. El plan estratégico de Wikimedia incluye la colocación de más sitios como cache del tráfico de América Latina, Asia y Oriente próximo.
- **codfw** (CyrusOne en Carrollton, Texas). Soporte de emergencia.

La infraestructura que permite tanto la comunicación interna entre esos centros, como la interacción con los clientes de internet se muestra en el siguiente esquema[34]:



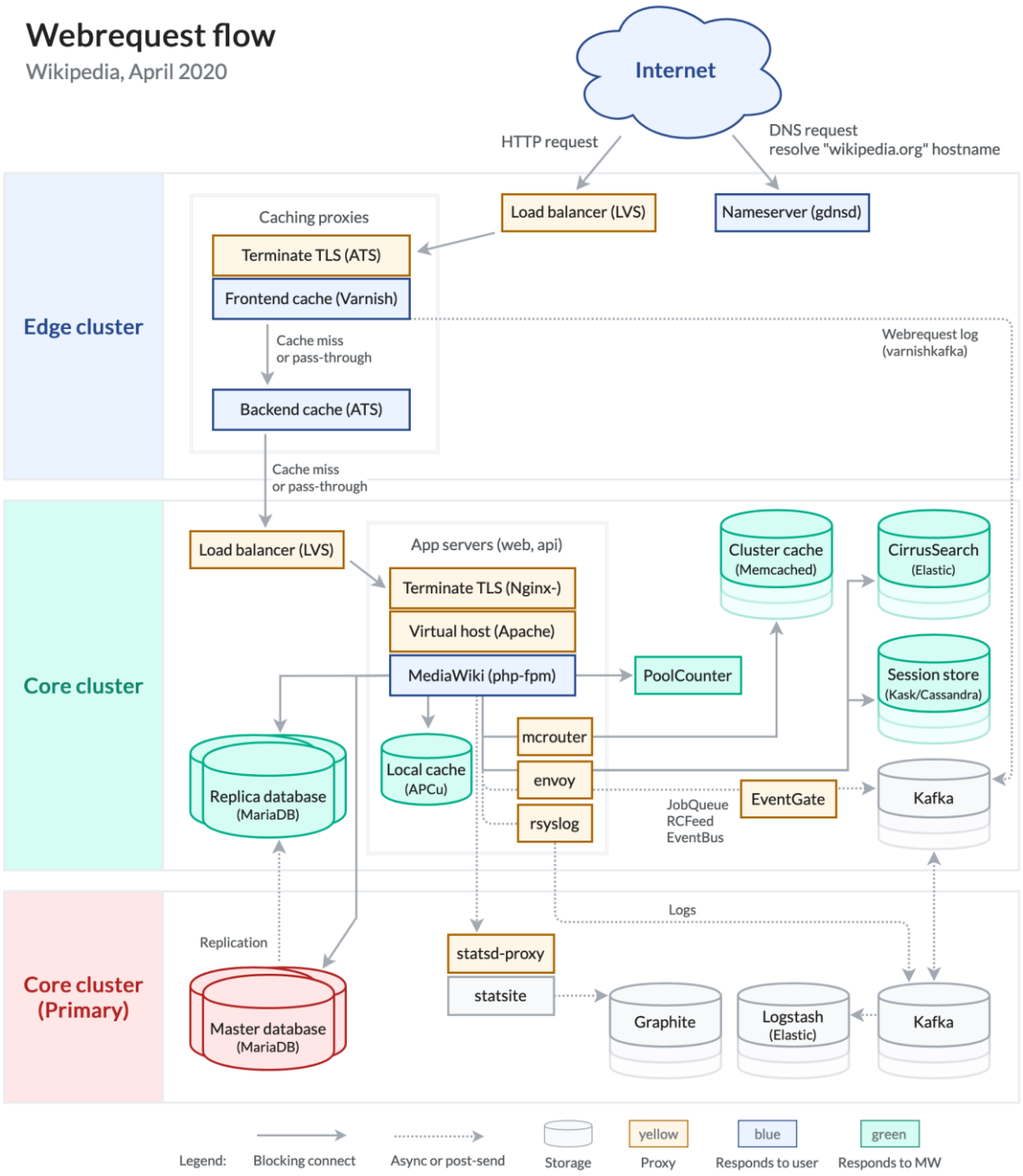
Cada uno de los nodos dispone de 3 ó 4 proveedores de acceso a la internet “genérica” (por la que llegan las peticiones de los clientes más “próximos”) junto a otras redes de carácter interno (para la comunicación con otros nodos pertenecientes a Wikimedia). Es fácil apreciar la heterogeneidad y complejidad de un sistema desplegado a escala mundial.

- **Componentes**[35]

Los componentes forman una pila LAMP muy compleja, adaptada para una instalación extrema:

[33] Más detalles en https://wikitech.wikimedia.org/wiki/Network_design
[34] En 2022 se añadió otro nodo en Marsella
[35] https://meta.wikimedia.org/wiki/Wikimedia_servers

- Wikipedia usa *gdnsd* para distribuir geográficamente las solicitudes entre los cinco centro de datos (3 en EEUU, 1 en Europa y 1 en Asia), dependiendo de la ubicación del cliente.
- Con Linux Virtual Server (*LVS*) se equilibra la carga de peticiones entrantes. LVS también se emplea para repartir la carga interna de solicitudes MediaWiki. La monitorización interna corre a cargo de un sistema propio (*PyBal*).
- Para agilizar las peticiones web habituales (artículos y API) se usan servidores proxy inverso *Apache Traffic Server* y *Varnish* delante de servidores *APACHE*.
- Todos los servidores ejecutan *Debian GNU/Linux*.
- Los objetos distribuidos se almacenan mediante *Swift*.
- La principal aplicación web es *MediaWiki*, escrita en PHP (~70 %) y JavaScript (~30 %)
- Los datos estructurados se almacenan en *MariaDB* desde 2013. Las wikis se agrupan en clusters, y cada cluster es atendido por varios servidores MariaDB, replicados en una configuración de maestro único.
- Para caching interno de queries a la BD y resultados de procesamiento se emplea *memcached*.
- Para búsquedas de texto en contenidos se emplea *ElasticSearch* (Extension:CirrusSearch)



Nuestra concepción de un sistema distribuido se asemeja mucho más a este último esquema que a los relacionados con la organización física detallada del sistema, con cientos o miles de equipos y conexiones de red. Un sistema distribuido depende más de la variedad de componentes que de la cantidad.

7 CONCLUSIONES

En la actualidad los sistemas de información son sistemas en red. Todos los casos que hemos visto anteriormente forman parte de los sistemas software conocidos como *Sistemas Distribuidos*.

Un diseño y desarrollo adecuados requieren un profundo conocimiento de la programación concurrente y de las arquitecturas utilizadas.

En los casos vistos anteriormente se muestran algunos aspectos importantes que guían los diseños de los sistemas distribuidos.

1. Flexibilidad en la funcionalidad. Un sistema se crea con unos requerimientos en un momento dado, pero dichos requerimientos varían con el tiempo.
2. Robustez. El sistema debe ser capaz de soportar fallos, manteniendo la integridad de la información, continuando con su función.
3. Rendimiento. El sistema debe tener unas características de rendimiento predecibles y adecuadas a su función. Normalmente el rendimiento se cuantifica en función de medidas de *throughput* (número de peticiones servidas por unidad de tiempo) y latencia (tardanza en completar una petición).
4. Escalabilidad. El diseño del sistema debe permitir adaptarlo a condiciones cambiantes de carga sobre el mismo, sin tener que realizar un rediseño, o alterar su implementación, mientras mantiene las características de rendimiento esperadas. Por tanto se trata de adaptabilidad dinámica.
5. Simplicidad. Simplicidad en la construcción, gestión de despliegue y la configuración del mismo. Esto incluye simplicidad en la reconfiguración.
6. Elasticidad. Capacidad de escalar un servicio, sin que sus usuarios noten interrupción o degradación alguna en el mismo, adaptando el conjunto de recursos utilizados a la carga existente en cada momento. El objetivo de la elasticidad es la obtención de escalabilidad minimizando el consumo de recursos y, por tanto, su coste.

Hemos considerado la computación en la nube como última etapa importante en la evolución de la computación

- Caracterizada por la eficiencia en el uso de los recursos
- Con un modelo de acceso de “pago por uso”
- Elasticidad es el objetivo principal

Una propiedad implícita que no es aparente en los casos de uso presentados es la relacionada con el uso apropiado de un sistema: la *seguridad*. En todos los casos presentados, la seguridad en varios de sus aspectos es de primordial importancia, condicionando en muchos casos las tecnologías elegidas para implantar la funcionalidad de la aplicación.

Las decisiones de arquitectura de un sistema incidirán en las propiedades de seguridad que se pueden conseguir en el sistema resultante, pero en esta asignatura no le podemos prestar la atención que le corresponde.

Hemos tomado la Wikipedia como servicio de escala mundial a través del que identificar los problemas y sus soluciones. Como las soluciones no son perfectas, provocan a su vez nuevos problemas muy específicos[36] del ámbito que nos ocupa.

Dentro del caso de estudio se citan y describen someramente múltiples ejemplos de otros servicios distribuidos actuales, utilizados por millones de usuarios. Sus arquitecturas exigen un diseño muy inteligente y cuidadoso para hacer frente a condiciones y magnitudes excepcionales.

Como sistema en evolución, las soluciones que se desarrollan van cambiando a lo largo del tiempo, de la misma manera que lo hacen las tecnologías y recursos de los que se dispone. Ningún sistema de esta escala puede permanecer inalterable en el tiempo porque la adaptación es un requisito imprescindible.

Las cifras manejadas en el caso de estudio pretenden acercarnos la realidad sobre la magnitud de los problemas. Disponer de recursos ilimitados, además de irreal, no es por sí mismo una solución: se necesita aplicar inteligencia para que la suma de recursos provoque una suma de capacidades y propiedades en el sistema final.

En general, para prestar sus servicios se han de repartir las solicitudes entre todos los nodos posibles, de manera que cada uno se responsabilice de una fracción de estas peticiones. Por otro lado se ha estudiado que la replicación de nodos posibilita la continuidad del servicio, y que existen técnicas específicas (caching, proxies inversos) para mejorar la capacidad de servicio.

Se habrá observado la necesidad de diseccionar el funcionamiento interno de la Wikipedia para encontrar formas de mejorar sus características como sistema distribuido. La identificación de componentes es una parte crucial en el diseño de los sistemas estudiados en TSR.

TSR es un curso de arquitectura de sistemas distribuidos. Su objeto central es el análisis de las alternativas de diseño en la especificación de dicho tipo de sistemas, considerando las tecnologías disponibles, y analizando su uso bajo el prisma de los aspectos citados anteriormente.

[36] Y, con suerte, de menor envergadura