## Índice

# Tema 1. Introducción

## Apuntes de Sistemas Distribuidos

Curso 2022-2023

# 1 Definición de Sistema Distribuido

Un libro clásico que estudia los Sistemas Distribuidos (SD) y su problemática es el de **Tanembaum**. Este autor propone la siguiente definición:

> Conjunto de computadores independientes que ante los usuarios del sistema parecen un solo computador

Esta definición plantea un aspecto clave de los SD, y es que están compuestos por varios computadores *independientes*, esto es, que cada uno de ellos por separado es un computador completo y autónomo, con su propia CPU (o conjunto de ellas si se trata de una arquitectura moderna multinúcleo), su propia memoria RAM y su propio subsistema de entrada/salida que puede (y suele) incluir un medio de almacenamiento externo, sea disco duro o memoria de estado sólido (flash).

Pero un SD no es meramente un conjunto de computadores, sino que estos además "*ante los usuarios del sistema parecen un solo computador*". Este requisito implica que los computadores han de ser capaces de interactuar entre sí de alguna forma para coordinarse y así lograr ese objetivo de "parecer uno solo". Es decir, implícitamente esta definición añade una *red de comunicación* al conjunto de computadores, así como un *software* capaz de coordinarlos para parecer uno solo.

El objetivo de "parecer uno solo", también denominado objetivo de *transparencia total* es demasiado restrictivo. Si siguiéramos esta definición, pocos SD tendríamos hoy día, salvo quizás algunos *clusters* (más adelante veremos qué son).

Desde esta definición original Tanembaum ha relajado este requisito y en su última edición del libro admite que hay casos en los que un conjunto de computadores interconectado y colaborando de alguna forma, puede no "parecer uno solo" y aún así seguir siendo útil y considerarse un sistema distribuido, o bien puede "parecer uno solo" a efectos de cierto servicio, pero no a todos los efectos.

Otro libro clásico en el campo es el de **Coulouris**, el cual da la siguiente definición para un SD:

> Conjunto de computadores autónomos, unidos por una red, que ejecutan un software diseñado para dar una utilidad computacional integrada

Observar cómo en esta definición la red ya se hace explícita, así como el *software* de interconexión. Y ya no se requiere que el conjunto "parezca un solo computador", sino que simplemente esté proporcionando una utilidad computacional integrada. Es decir, que entre todos proporcionen una utilidad al usuario.

En su última edición, Coulouris ha dado una definición aún más general:

> Un sistema distribuido es aquel en el que los componentes hardware o software localizados en computadores unidos mediante red, comunican y coordinan sus acciones *sólo mediante el paso de mensajes*.

Ya que estas definiciones pueden resultar un tanto abstractas, veremos seguidamente algunos ejemplos de lo que se puede considerar un sistema distribuido.

# 2 Ejemplos de sistemas distribuidos

## 2.1 Sistemas de archivos distribuidos

Cuando la cantidad de datos que se necesitan tener almacenados en disco excede la capacidad de un único ordenador (como suele ocurrir con el llamado *big-data*), pueden usarse varios ordenadores cooperando entre sí, cada uno con su disco, para producir la ilusión de que se trata de un solo disco mucho más grande. Un solo fichero enorme podría estar distribuido en varios discos, de forma transparente al usuario que lo vería como si fuera un solo objeto continuo almacenado de alguna forma en su ordenador local. A la vez, la existencia de múltiples ordenadores permitiría *replicar* información, es decir, que los diferentes trozos del fichero estén almacenados en más de un computador. De este modo, aún si algunos de ellos fallan, el fichero seguiría siendo accesible.

Un ejemplo de sistema de archivos de estas caraceterísticas es HDFS (Hadoop File System).

## 2.2 Aplicaciones P2P para compartir archivos

Son aplicaciones distribuidas compuestas por varios nodos cada uno de los cuales está ejecutando un proceso que es el mismo en todos ellos. Es decir, a diferencia del Web en el que hay claramente dos tipos de procesos (el cliente y el servidor), en las aplicaciones P2P no hay clientes ni servidores, sino que todos los procesos desempeñan ambos roles a la vez.

Las aplicaciones P2P permiten la diseminación de contenidos (documentos o cualquier otro tipo de archivos) sin la necesidad de un nodo central que los almacene y que a la larga podría suponer un cuello de botella si muchos clientes intentan la descarga simultáneamente. En una aplicación P2P el documento compartido está repartido entre muchas máquinas, cada una de las cuales puede almacenar sólo un trozo del recurso, o bien el recurso completo, y puede actuar como servidor para suministrar el trozo correspondiente.

Un ejemplo de este tipo de aplicación es el protocolo *bittorrent*.

## 2.3 Computación en malla (*grid computing*)

La idea de la computación en malla parte de una analogía entre los computadores y las centrales generadoras de energía eléctrica. El suministro de energía se organiza en una red o malla a la que se conectan las diferentes centrales eléctricas, que generan la energía, y también las fábricas y domicilios particulares que la consumen. Alguna de estas fábricas o domicilios puede a su vez ser generador de energía (por ejemplo viviendas que tengan instados paneles solares), y contribuyen por tanto en inyectar energía a esa malla que pueda ser consumida en otra parte.

La idea del *grid computing* es algo similar. Los computadores pueden verse como recursos capaces de ofrecer *capacidad de cómputo*, a la vez que habrá otros que sean clientes que necesiten realizar complejos cálculos que demanden mucha capacidad de cómputo. Se trata de organizar todo ese parque de millones de computadores que muchas veces están ociosos para que contribuyan con su capacidad de cómputo a las necesidades de cómputo que pueda existir en otra parte. Por ejemplo, instituciones de investigación de astronomía, física, meteorología, medicina, etc. a menudo tienen grandes necesidades computacionales y tradicionalmente las resolvían mediante la compra o alquiler de grandes supercomputadores. La computación en malla vendría a mejorar la situación al permitir que varias de estas instituciones colaboren "cediéndose" capacidad de cómputo cuando no la necesitan, o usando incluso la de los ordenadores de sobremesa de los voluntarios que quieran colaborar.

Evidentemente la coordinación de esos computadores requerirá de un software intermedio que gestione cuáles de ellos forman parte de la malla, qué capacidad de cómputo pueden ofrecer, si están libres u ocupados, etc. Este software intermedio sería un *middleware grid*, y a día de hoy existen muchas implementaciones (por ejemplo Globus, BOINC)

Sobre este *middleware grid* es posible construir *aplicaciones grid* que son las que finalmente implementen la solución a un problema dado que requiera gran capacidad de cómputo. Por ejemplo, sobre el middleware BOINC se desarrolla todo un conjunto de aplicaciones grid de computación voluntaria, tales como SETI@home para la búsqueda de señales inteligentes procedentes del espacio exterior, lo que requiere ejecutar complejos algoritmos sobre las señales registradas por los radio-telescopios, LHC@home (también llamado Sixtrack) que colabora con el CERN en el análisis de los resultados obtenidos de su acelerador de partículas, Rosetta@home que ejecuta algoritmos que

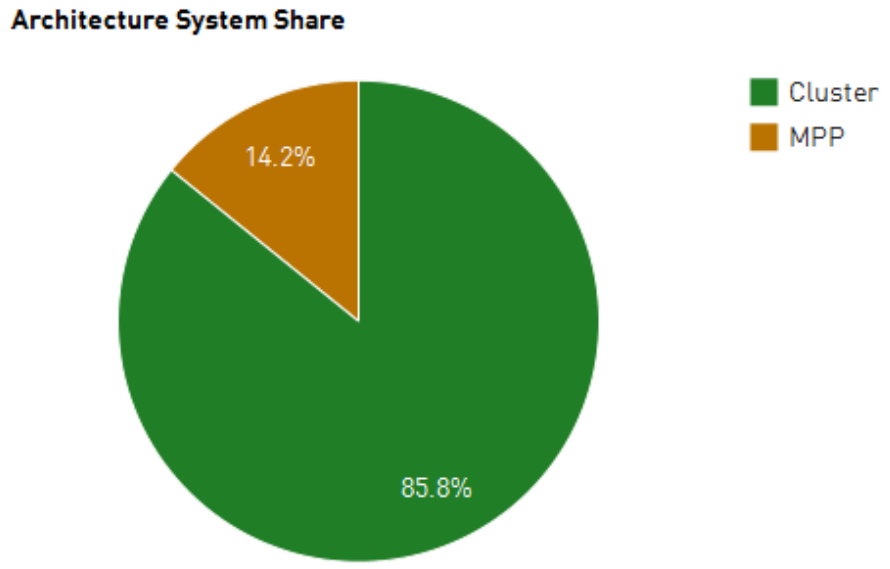
determinan los posibles plegados tridimensionales de las complejas moléculas de proteínas, con aplicaciones en biología, medicina y farmacología, y muchos más.

## 2.4 Clusters

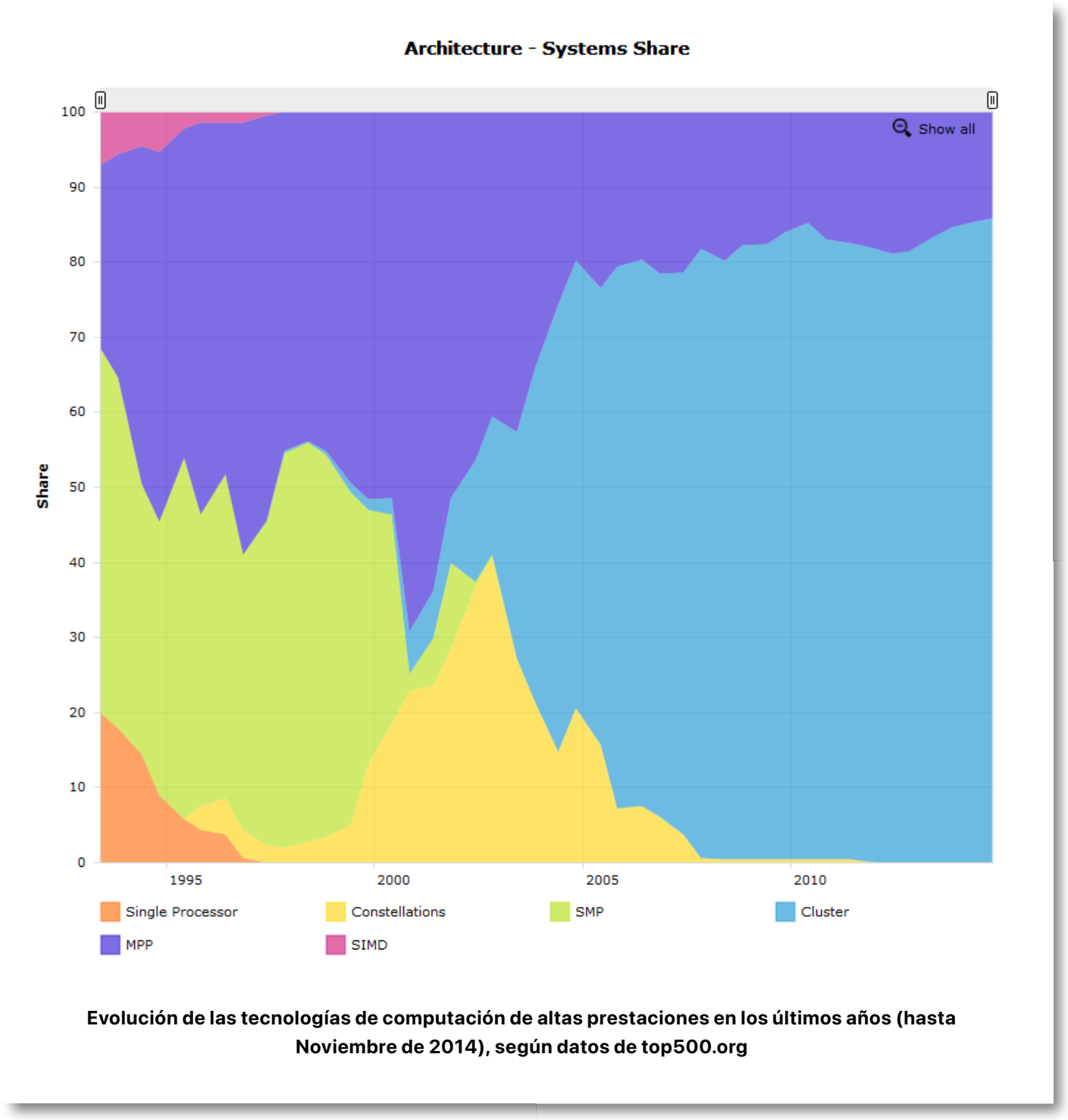
Un *cluster* es un sistema distribuido compuesto por ordenadores *homogéneos* (es decir, que tienen la misma arquitectura, el mismo tipo de CPU) y normalmente próximos geográficamente (generalmente montados en un mismo armario en diferentes *racks*), conectados mediante una red de alta velocidad, y en los que se ejecuta un software que les hace estar muy interrelacionados hasta el punto de que para el usuario que accede a ellos desde una terminal parece un único computador.

Los *clusters* se construyen como una alternativa y mucho más barata a los *supercomputadores*. El cluster tiene la ventaja evidente de que puede crecer (añadiendo más computadores) si en algún momento su capacidad de cálculo o de memoria se queda pequeña, algo mucho más difícil y costoso de lograr en un supercomputador.

A día de hoy muchos grandes centros de datos se construyen mediante *clusters*, como por ejemplo los utilizados por Google, o por Amazon, aunque de éste se sabe menos. En la figura puede verse cómo la tecnología *cluster* domina sobre otras que en el pasado tuvieron más protagonismo, como la *MPP* (*Massively Parallel Computing*).




**Reparto de tecnologías más usadas, a Noviembre de 2014, según top500.org**




**Evolución de las tecnologías de computación de altas prestaciones en los últimos años (hasta Noviembre de 2014), según datos de top500.org**

Algunos de ellos usan *software propietario*, desarrollado por la propia compañía, para lograr la funcionalidad que buscan en el cluster. Otros usan sistemas operativos específicos para clusters (que suelen ser una variante de Linux), como es el caso de los clusters Beowulf.

Esta conjunción de hardware y software ofrece como resultado una solución de computación paralela de altas prestaciones, a partir de elementos asequibles (la infraestructura de Google está montada a base de PCs de gama media-alta). Sobre esta plataforma se puede ejecutar cualquier aplicación que requiera de computación intensiva.

Muchas de las compañías que tienen un *cluster*, o varios de ellos separados geográficamente pero conectados entre sí, para sus necesidades de computación, tienen la mayor parte del tiempo excedente de capacidad y están en disposición de *alquilar* esa capacidad sobrante para que otras compañías o particulares puedan hacer uso de ella. Esta tendencia se conoce como *cloud computing*.

## 2.5 Cloud computing

Si, sobre los recursos hardware de un *cluster*, se despliga un servicio de virtualización que permita crear máquinas virtuales de forma automatizada y sin necesidad de intervención humana, de modo que los clientes puedan directamente crear y arrancar máquinas virtuales que se ejecuten en el *cluster*, sin necesidad de conocer qué hardware en particular lo está ejecutando, y posibilitando incluso que un mismo hardware sea compartido por varios clientes, cada uno ejecutando su máquina virtual, se habla de "Infrastructure as a Service" (o IaaS), una de las posibilidades del cloud computing.

El propietario del hadware en que se ejecutan las máquinas virtuales puede monitorizar en todo momento los recursos que éstas están utilizando, y cobrar al cliente en función de los recursos usados. Esto tiene muchas ventajas de cara al cliente que no necesita comprar ni mantener su propio hardware, y también de cara al proveedor, que puede obtener beneficio de recursos no utilizados.

Sobre un *cloud* es posible desplegar múltiples máquinas virtuales, unirlas mediante una red virtual, y crear así todo tipo de sistemas distribuidos, tanto para almacenamiento de datos como para ejecución de algoritmos distribuidos. Una típica aplicación web puede desplegarse en múltiples máquinas (por ejemplo, una para servir ficheros estáticos usando HTTP, otras para atender peticiones dinámicas que conlleven la ejecución de código, otras para mantener una base de datos distribuida, etc.)

Para el usuario de la aplicación web no habrá diferencia entre esta arquitectura *orientada a servicios* o una aplicación monolítica en la que todo se está ejecutando en un solo proceso. Sin embargo para los desarrolladores de la aplicación la diferencia es fundamental, ya que la arquitactura orientada a servicios le permite añadir o quitar recursos de los servicios que lo necesiten, de forma independiente de otros servicios, replicar los servicios críticos para que no dejen de estar disponibles ante fallos, desarrollar y actualizar por separado cada servicio y desplegarlo sin tener que detener la aplicación (integración continua), etc.

# 3 Ventajas de los Sistemas Distribuidos

¿Por qué se construyen los sistemas distribuidos? En muchas ocasiones la razón es que se desean compartir recursos que están *físicamente* distribuidos. Por ejemplo, dos o más ordenadores situados en diferentes plantas de un edificio de oficinas desean compartir una impresora, situada en otra planta. Aquí los recursos están físicamente distribuidos, y no cabe plantear como "solución" el reunirlos (es decir, llevar a una misma habitación todos esos ordenadores y la impresora). Por tanto la solución

consiste en conectar entre sí estos elementos con una red y escribir un *software* que les permita compartir el recurso en cuestión. Es decir, construir un sistema distribuido.

Son muchos los ejemplos que podrían ponerse de sistemas distribuidos construidos a partir de este problema: la red de cajeros de un banco, la intranet que conecta todas las sucursales de una franquicia, una red de vigilancia compuesta por varios PCs conectados con cámaras de vídeo y un PC "central" que recibe imágenes de todos ellos, una red de sensores similar en una industria (suponiendo que cada sensor lleva parejo un computador capaz de hacer un procesamiento más o menos sofisticado de la señal que recibe, y que lo que se pretende es mostrar todos los resultados de forma conjunta en una sola terminal), etc.

La propia Internet es un recurso distribuido, compuesto por todos los ordenadores que están conectados a ella. Muchos de esos ordenadores son dispositivos móviles, lo que hace aún más evidente su naturaleza distribuida.

En otras ocasiones, como en el caso del *grid computing* o de las redes *bittorrent* el recurso que se comparte es algo más abstracto, no algo tan físico como una impresora, una cámara o un sensor, sino una capacidad como el espacio de almacenamiento en disco el uso de una CPU, o el ancho de banda de una red. Aunque el recurso es más abstracto, sigue siendo cierto que está *físicamente* distribuido y que no es "solución" el juntar físicamente esos recursos.

Finalmente hay casos en los que los elementos que componen el sistema distribuido **no** están físicamente distribuidos, o no hay necesidad de que lo estén, como es el caso de los *cluster*. En este caso ¿por qué se construye un sistema distribuido?

Estas son las razones más importantes:

- **Coste** Un *cluster* puede construirse con PCs baratos. Aunque, para lograr la capacidad computacional deseada, puede ser necesario juntar miles de ellos, el coste es aún menor que el que tendría un *supercomputador* del mismo rendimiento. Es decir, el factor coste/rendimiento es muy bueno.

- **Escalabilidad** Esta propiedad hace referencia a la posibilidad de *crecer*. Si la capacidad computacional se queda pequeña porque, por ejemplo, crece la cantidad de datos que el sistema tiene que manejar, o la complejidad de los algoritmos que tiene que ejecutar, siempre es posible aumentar esa capacidad añadiendo más computadores al sistema. Aunque hay que tener en cuenta que a mayor número de computadores en el sistema, mayor cantidad de mensajes se tendrán que intercambiar, por lo que la red puede llegar a convertirse en un cuello de botella que impida una escalabilidad ilimitada.

- **Rendimiento** La velocidad de procesamiento de un sistema distribuido depende de la cantidad de ordenadores que lo componen, puesto que todos ellos pueden estar procesando en paralelo. Naturalmente los algoritmos han de estar diseñados de forma que aprovechen al máximo la

información local de la que dispone cada ordenador, minimizando las necesidades de comunicación y sincronización con los demás.

- **Tolerancia a fallos** Si definimos el *factor de disponibilidad* (en inglés *availabilty*) como la fracción del tiempo que un sistema está *disponible* (esto es, puede ser usado), con respecto al tiempo total (la suma del tiempo en que está disponible más el tiempo en que no está disponible) tendremos una medida de cuán tolerante a fallos es el sistema.

  Por ejemplo, si un sistema tiene un fallo aproximadamente cada 500 días, y la reparación del fallo lleva un tiempo de tres horas (durante las cuales el sistema no es utilizable) ¿Cuál es su factor de disponibilidad?

  Para encontrar la respuesta pasamos a horas el tiempo entre fallos, lo que resulta en 500×24 = 12000 horas, por lo que el factor de disponibilidad será 12000/(12003), es decir 0.999750062 que es un 99.975%.

  ¿Es mayor la disponibilidad de un sistema distribuido o la de un sistema centralizado? La respuesta a esto puede ser un arma de doble filo. En un sistema compuesto por 1000 computadores, la probabilidad de que uno en concreto falle puede ser muy baja. Pero la probabilidad de que falle *cualquiera* de ellos es mucho mayor. Por tanto la distancia entre fallos en el sistema distribuido cabe esperar que será mucho menor que la de un sistema centralizado, es decir, que tenga fallos más a menudo. Más aún si además tenemos en cuenta que la red es otro elemento que puede fallar en cualquiera de sus puntos, y que la topología de red suele ser muy compleja.

  Si bien lo anterior es cierto, la clave está en darse cuenta de que en un sistema centralizado, un fallo del sistema equivale a sistema inutilizable, y por tanto no disponible mientras se repara, pero en un sistema distribuido esto no tiene por qué ser así.

  Si el sistema distribuido está bien diseñado, y contiene *redundancia*, un fallo en una de sus partes puede ser asumido por las que aún funcionan, y por tanto el sistema puede seguir estando disponible, *incluso mientras está siendo reparado*. Por tanto un buen diseño llevaría a una disponibilidad mucho más alta que la de un sistema centralizado.

  Por desgracia el diseño de sistemas distribuidos es un asunto complejo, y así no es de extrañar la humorística definición de sistema distribuido atribuida a Leslie Lamport:

  > Sabes que estás ante uno [sistema distribuido] cuando el fallo de un computador que ni siquiera sabías que existía, te impide a ti terminar tu trabajo.

Las principales dificultades en el diseño de los sistemas distribuidos se centran básicamente en este asunto de la tolerancia a fallos, como veremos. Los fallos ocurrirán, y el sistema debe estar diseñado de tal forma que pueda seguir funcionando pese a ello. Esto es sorprendentemente difícil de conseguir, e incluso puede ser imposible según como definamos *fallo* y el tipo de red que

usemos para conectar los computadores.

# 4 Metas y retos de los Sistemas Distribuidos

Hay una serie de características deseables en los sistemas distribuidos, pero cuya consecución es tecnológicamente compleja. Por esa razón las consideramos como metas a perseguir, a la vez que retos.

Estas metas están muy relacionadas con las ventajas que esperamos de los sistemas distribuidos, las cuales se pueden convertir en inconvenientes si no se resuelven adecuadamente. Así, por ejemplo, hemos visto que una ventaja importante de los SD es que permiten compartir recursos entre diferentes usuarios. Esto conlleva inmediatamente una serie de nuevos problemas: ¿qué usuarios pueden acceder a qué recursos? ¿cómo garantizar que el usuario que intenta acceder tiene efectivamente el derecho de hacerlo? ¿cómo verificar la identidad de los usuarios? ¿cómo garantizar la privacidad de las comunicaciones? Son todos aspectos relacionados con la *seguridad*. También hay aspectos de usabilidad. Se espera que los usuarios sean capaces de manejar el sistema sin necesidad de conocer detalles de su arquitectura o de la distribución física de los recursos que comparten. Son aspectos de *transparencia*. El mundo de la informática es muy dinámico, continuamente aparecen aplicaciones y protocolos nuevos. Para el arquitecto del sistema distribuido resulta importante que el añadido de nuevas funcionalidades o su interconexión con otros sistemas sea lo más simple posible. Son aspectos de *modularidad* y *estandarización*. Finalmente la mayoría de los SD crecen más allá de lo que inicialmente habían sido estimados. Es importante que su arquitectura permita este crecimiento. Son aspectos de *escalabilidad*.

Así pues los retos más importantes son justamente estos (y dedicaremos temas específicos dentro de esta asignatura a varios de ellos):

- **Seguridad** que engloba muchos aspectos diferentes, relacionados principalmente con la garantía de la **privacidad** de las comunicaciones (que una tercera persona no pueda averiguar su contenido si no está autorizada a ello), su **integridad** (que una tercera persona no pueda alterar un mensaje sin conocimiento de los comunicantes) y su **autenticidad** (que los comunicantes tengan garantías de realmente con quien se están comunicando es con quien ellos creen y no con una tercera persona que se hace pasar por él). La mayoría de estos problemas se resuelven tecnológicamente con algoritmos criptográficos de clave asimétrica. Veremos algo de esto en los temas finales. No obstante hay una importante componente social, de *educación* de los usuarios, que está aún por resolver.

  La seguridad incluye también otros aspectos más relacionados con la calidad del código, que puede presentar *bugs* que se conviertan en *agujeros* a través de los cuales una código pueda adquirir permisos para operaciones que no le corresponden, o incluso permisos de administración del sistema, lo que le permitiría cambiar código en el mismo, certificados, etc. comprometiendo

seriamente toda la seguridad lograda por los algoritmos criptográficos. Un sistema distribuido, al estar necesariamente conectado a una red, es vulnerable a este tipo de *ataques* que comprometen su **disponibilidad**. Un reto de diseño es protegerlo frente a estos ataques.

- **Transparencia** Esta meta está relacionada con la *ocultación de la naturaleza distribuida* del sistema. Cuanto más se parezca un sistema distribuido a uno centralizado, en lo que respecta a su uso o su programación, mayor grado de transparencia se habrá logrado. El usuario debería usar ese sistema de tal forma que no pueda distinguir si "al otro lado" hay una red de máquinas interconectadas, o una sola.

  Esta meta tan ambiciosa puede descomponerse en una serie de etapas más o menos progresivas:

  - **Transparencia de acceso**. Sería el nivel más básico en el que se oculta al usuario la heterogeneidad de la información (que las diferentes máquinas utilizan diferentes códigos para representar los mismos datos), de la arquitectura (que las diferentes máquinas usan diferentes CPUs, diferentes mecanismos de acceso a memoria, etc.), del Sistema Operativo, y del lenguaje en que ha sido programada cada parte del SD. El usuario aún percibe el sistema como compuesto por varias máquinas interconectadas, pero accede a los servicios de todas ellas en la misma forma, "parecen iguales".
  - **Transparencia de Localización**. Se oculta qué máquina tiene qué recurso. Por ejemplo los servicios "en la nube" tipo Dropbox u OneDrive, que almacenan ficheros del usuario. Esos ficheros están realmente almacenados en alguna máquina física (quizás repartidos entre varias), pero el usuario no tiene noción de que así sea. Parecen estar todos juntos en algún tipo de disco virtual.
  - **Transparencia de Replicación**: oculta si un recurso está repetido para mejorar la fiabilidad, como de hecho suele ser el caso.
  - **Transparencia de Migración**: oculta si un recurso se mueve de una máquina a otra. Se entiende que esto está íntimamente ligado a la transparencia de localización, ya que si no existe transparencia en la localización (el usuario es consciente en qué máquina están los recursos), tampoco existirá transparencia en la migración (será consciente de cuándo se mueven de una máquina a otra). Pero no basta con la transparencia de localización para lograr la de migración. El usuario no debe percibir esa migración tampoco mientras está teniendo lugar.
  - **Transparencia ante Fallos**: oculta si hay un fallo en una máquina del SD (el resto se hacen cargo de las funciones de la máquina caída). Esta meta plantea exactamente la situación contraria a aquella de la que se lamentaba Lamport en su definición humorística. Parece claro que para lograr la transparencia ante fallos ha de haber también una de migración o replicación, o ambas, lo que implica también una de localización.

  El objetivo ideal de que el sistema parezca ante el usuario "como una sola máquina", siempre fiable y sin fallos, no siempre es lo más deseable. Hay casos en los que, más bien al revés, resulta conveniente hacer patente la distribución física de los componentes y los recursos. Por ejemplo, en

sistemas de *computación ubicua* en la que muchos de los nodos del sistema son dispositivos móviles, puede ser necesario conocer dónde está cada uno en cada momento, de cara a aprovechar *oportunidades* cuando uno de ellos está cerca de un recurso que interese. Sería el campo de las *redes oportunistas*.

- **Apertura** Esta meta (en inglés *openness*) se centra en el uso de estándares, el diseño modular, basado en componentes, el código abierto o al menos los interfaces abiertos, etc. Esta apertura es muy beneficiosa en el ámbito de los Sistemas Distribuidos porque facilita la *interoperabilidad* de estos sistemas, y por tanto la capacidad de un SD de ofrecer nuevos servicios mediante la agregación de otros SD.

  Un claro ejemplo del éxito de esta vía es el *World Wide Web*, una de las aplicaciones más exitosas de Internet (de hecho, mucha gente cree equivocadamente que el WWW *es* Internet). Su expansión y crecimiento ha sido posible gracias al uso de protocolos abiertos (HTTP) y lenguajes abiertos para la codificación de la información (HTML, XML), rápidamente estandarizados por el W3C, que han posibilitado la implementación de funcionalidades no imaginadas en su origen. La apertura es la clave para la *extensibilidad*.

- **Escalabilidad**. Esta meta hace referencia a la posibilidad de crecer de un Sistema Distribuido (cambiar de *escala*). El sistema debe ser capaz de adaptarse al tamaño de los datos que tiene que manejar. Si no hay suficiente capacidad de cómputo o de almacenamiento, debería ser posible una solución mediante la adición de más computadores. Sin embargo para que esto sea posible, el diseño ha de ser escalable ya desde su origen.

  A menudo un mal diseño inicial compromete la escalabilidad. Por ejemplo, si cada nodo del sistema es identificado mediante un número de 8 bits, esto implica que nunca podrán añadirse más de 256 nodos, y por tanto se tiene un límite al posible crecimiento. En realidad este ejemplo no es muy diferente de lo ocurrido con las direcciones IPv4, formadas por 32 bits. Varias soluciones imaginativas como el uso de redes locales y *Network Address Translation* (NAT) consiguieron extender un poco más la vida de este tipo de direcciones, pero ha sido ya necesario rediseñar el protocolo IP (IPv6) para dar cabida a direcciones con 128 bits.

  El anterior es un ejemplo de limitación de la escalabilidad por un recurso *software*. Obviamente también los recursos *hardware* pueden comprometer la escalabilidad. Por ejemplo, si existe un servidor central que tiene que participar en todas las transacciones que tienen lugar en el SD, está claro que este servidor estará marcando un límite a la escalabilidad, ya que a partir de un cierto número de nodos no tendrá capacidad suficiente como para atenderlos a todos. Ese nodo sería un *cuello de botella*. La propia red puede convertirse en un cuello de botella, más tarde o más temprano dependiendo bastante de su topología.

  La clave para mantener la escalabilidad lo más alta posible consiste en diseñar algoritmos y soluciones *descentralizados*, es decir, que puedan funcionar en ausencia de un servidor central. El

protocolo de enrutamiento IP es un ejemplo excelente de algoritmo descentralizado, y a él se debe que la Internet pueda existir con el inimaginable número de nodos con que cuenta actualmente. El sistema de servidores de nombres (DNS) usado en Internet es otro buen ejemplo de protocolo descentralizado. Si todas las peticiones de DNS tuvieran que pasar por un servidor central, ese *cuello de botella* tendría que ser sustituido cada poco por uno de mayor potencia, comprometiendo el crecimiento de Internet.

Como se desprende de lo anterior, diseñar e implementar sistemas distribuidos es una tarea muy compleja. De hecho esta puede considerarse su principal *desventaja* con respecto a una solución no distribuida. Todos los retos anteriores, si no están correctamente resueltos, son puntos en contra. Un sistema distribuido puede fallar de incontables nuevas formas que no tenemos que considerar en un sistema centralizado.

La que sigue es una famosa lista de *falacias* (suposiciones incorrectas) que a menudo hacen los programadores de aplicaciones distribuidas y que, al no ser ciertas, tarde o temprano causarán problemas. La lista fue escrita por Peter Deutsch[1]:

1. La red es fiable
2. La latencia es cero
3. El ancho de banda es infinito
4. La red es segura
5. La topología no cambia
6. Hay un administrador de la red
7. El coste del transporte es cero
8. La red es homogénea

# 5 Modelos o paradigmas

Como parte de este primer tema introductorio, daremos una visión general de algunos de los paradigmas más importantes en los que se basan las arquitecturas de los sistemas distribuidos. Algunos de ellos se tratarán con mucho más detalle en los temas posteriores.

## 5.1 Paso de mensajes

Este es el modelo más básico y fundamental de interacción entre dos componentes de un sistema distribuido. Es la base que subyace a todos los demás modelos, puesto que en el fondo todos los restantes se implementan sobre una capa de paso de mensajes.

En este modelo la interacción tiene lugar a través de dos primitivas básicas mediante las cuales dos

nodos o entidades pueden intercambiarse información.

- **Enviar** (*send*) Envía información desde el nodo local hacia el remoto
- **Recibir** (*recv*) Recibe la información que viene del nodo remoto al local

Dentro de este paradigma básico hay dos modalidades de interacción:

- **Orientada a conexión** (a veces también llamada orientada a flujo, o *stream*). En ella las dos entidades que van a comunicarse conectan previamente una con otra a través de un protocolo de *handshaking* y como resultado del mismo se crea entre ambas un canal virtual de comunicación que se comporta como una "tubería", de tal modo que toda la información que se "vierte" en un lado de ese canal, aparece al otro lado en el mismo orden en que se volcó.

  Una analogía típica usada para explicar este concepto es el de la comunicación telefónica. Dos personas que se comunican telefónicamente primero deben conectar mediante un protocolo que consiste en que uno de ellos efectúa una llamada (necesita conocer el número del otro), y el otro descuelga cuando recibe esa llamada. Tras esto, se ha creado entre ambos un canal virtual por el que circula información, y evidentemente las palabras llegan al extremo en el mismo orden en que fueron pronunciadas. El canal se mantiene hasta que uno de los extremos cuelga. El otro, al percibir este hecho, cuelga también y la comunicación termina.

  Para producir la ilusión de ese canal virtual sobre una red con una topología que puede ser muy compleja, es necesario un protocolo que por debajo descomponga cada envío de información en un número de paquetes, en el que cada uno de ellos lleva incrustada la dirección a la cual debe llegar, junto con el número que hacía cada paquete en la secuencia original, y estos paquetes son enviados a los nodos vecinos del emisor (los que están físicamente conectados con él), los cuales a su vez los reenvían a otros nodos, tomando decisiones de encaminamiento que aumenten la probabilidad de que cada paquete llegue eventualmente a su destino. Cuando cada paquete llega, el destino envía a su vez un paquete de retorno para señalar la correcta recepción. A la vez va ensamblando el mensaje original gracias al número de secuencia que cada paquete lleva. Si falta algún paquete intermedio y después de un tiempo razonable no ha llegado, el nodo receptor puede solicitar su reenvío. El emisor reenviará de todas formas más tarde aquellos paquetes para los cuales no haya recibido la confirmación. Si todo va bien, todos los paquetes acabarán llegando a su destino y el mensaje será reconstruido con lo que podrá ser entregado al proceso a quien iba dirigido.

  Vemos que la idea del *canal virtual* por el cual comunicarse es muy potente y simplifica bastante la programación de estas comunicaciones, pero a cambio el protocolo que va "por debajo" se hace más complejo.

- **Orientado a datagramas**. En este segundo modelo el emisor simplemente envía un datagrama al destinatario. Los protocolos subyacentes harán todo lo posible (*best effort*) por que el paquete

llegue a destino, usando algoritmos de encaminamiento que maximicen esa probabilidad, evitando nodos intermedios caídos o en los que los enlaces están saturados. Sin embargo, cuando el paquete llegue a destino, el protocolo de red no enviará un acuse de recibo por lo que el emisor nunca sabrá si realmente ha llegado, a menos que el proceso a quien iba dirigido envíe a su vez otro datagrama de respuesta.

Si el emisor tiene que enviar mucha información y toda ella no cabe en un solo datagrama, será el propio proceso emisor el encargado de romper esta información en paquetes más pequeños y mandar cada datagrama en un envío separado. El protocolo de red no da garantías de que lleguen en el mismo orden en que fueron enviados, ya que el algoritmo de encaminamiento puede elegir una ruta diferente para cada paquete, y por tanto cada uno tendrá una latencia diferente. El emisor tendría que incorporar números de secuencia en los datagramas si quiere que el receptor pueda reordenarlos correctamente (y el receptor obviamente estará programado para comprender y usar esos números).

Una analogía típica para comprender este modelo de comunicación es el sistema de correo postal ordinario. La unidad de información es la carta. El emisor se limita a depositarla en un buzón para que el sistema de correos se haga cargo de ella. Éste no da garantías de que la carta llegue a destino, aunque lo hará lo mejor que pueda para que llegue. Si queremos garantías, el destinatario deberá escribir otra carta de respuesta. Si enviamos varias cartas seguidas podrían llegar en diferente orden al que fueron enviadas.

Como vemos, este modelo viene a ser lo contrario del orientado a conexión. Y vemos cómo la programación de la comunicación se vuelve más compleja a cambio de que los protocolos de red se vuelvan más simples (y por tanto más rápidos)

Generalmente se prefiere un modelo orientado a datagramas cuando emisor y receptor se comunican a través de una red muy fiable, en la que la probabilidad de pérdidas de paquetes es baja. En este caso se logra la mayor velocidad de comunicación posible al aligerar los protocolos de red y los acuses de recibo que tienen que viajar por ella. Por ejemplo si ambos están próximos físicamente. En este caso también se reduce la posibilidad de que los datagramas lleguen desordenados, al ser menor el número de nodos intermedios que tienen que atravesar. También se usa la comunicación orientada a datagramas para el caso de que la pérdida o desorden de alguno de ellos no sea un impedimento a que el sistema siga funcionando. Por ejemplo, en el *streaming* de vídeo o audio, algún datagrama perdido se traduciría en la pérdida de algún *frame* de vídeo, en una momentánea parada de la imagen o un silencio o chasquido en el audio, pero en líneas generales la comunicación sería posible.

Por el contrario, si la red no es fiable (por ejemplo entre nodos separados por grandes distancias geográficas), y la aplicación no puede tolerar perder datos o recibirlos fuera de orden, se prefiere un modelo orientado a conexión aunque proporcione menor velocidad.

Estos modelos están implementados en la API de *sockets* que estudiaremos con más detalle en un

tema posterior. El modelo orientado a conexión es implementado por el protocolo TCP, mientras que el orientado a datagramas es implementado por el protocolo UDP.

## 5.2 Modelo cliente-servidor

La aplicación distribuida se compone de varios procesos que se comunican, y en este modelo cada un de los procesos desempeña uno de los siguientes roles:

- **Servidor**. Este rol es pasivo. El proceso está la mayor parte del tiempo inactivo (el sistema operativo lo pasará a estado bloqueado para ahorrar recursos), hasta que se produzca la llegada de un mensaje. Entonces el proceso continúa su ejecución, recibe el mensaje, lo procesa y lleva a cabo alguna acción especificada en ese mensaje. Generalmente se usa el término *petición* (*request*) para designar al mensaje recibido, y *servicio* a la acción realizada por el servidor en respuesta a esa petición. Lo más frecuente es que como resultado del servicio el servidor envíe una *respuesta* en forma de otro mensaje hacia el proceso que hizo la petición.

- **Cliente**. Este rol es activo. El cliente es un proceso que está ejecutando algún tipo de funcionalidad necesaria para el usuario, hasta que llega a un punto en que, para completar esa funcionalidad, necesita un servicio que sólo está disponible en una máquina remota. Entonces compone un mensaje con la *petición* y se lo envía al servidor. Tras esto puede quedarse inactivo (bloqueado) esperando por la respuesta, o bien puede continuar con otras tareas (cómputo, atención a la interfaz con el usuario, etc.) hasta que eventualmente recibirá la *respuesta* del servidor que le permitirá finalizar la funcionalidad pendiente.

Un gran número de aplicaciones de Internet (por no decir la gran mayoría) funcionan según este modelo. Por citar el ejemplo más sobresaliente, en el que probablemente estés ya pensando, tenemos el *World Wide Web* (WWW). Se trata de una red de documentos hiper-media (texto, gráficos, sonidos y enlaces de una parte del documento a otro y entre documentos) almacenados de forma distribuida en una red de servidores. El cliente es el navegador web, que se ocupa de realizar las peticiones de los documentos a los servidores apropiados (identificados a través de un *URL*) en respuesta a eventos desencadenados por el usuario (fundamentalmente la pulsación de los botones del ratón). El servidor es obviamente el servidor web, que realiza un servicio que habitualmente consiste en recuperar un documento de un medio de almacenamiento o de una base de datos, o bien componer ese documento "al vuelo" a partir de información extraída de una base de datos, y enviar ese documento como *respuesta* a la petición del cliente.

Pero hay muchos más servicios implementados de acuerdo con este modelo. Por ejemplo, las aplicaciones de correo electrónico (email) en las que el cliente es el programa que el usuario utiliza para leer o responder los mensajes, y el servidor es la máquina remota que se ocupa de almacenarlos a la espera de que el cliente los solicite y los descargue, y de hacer llegar los mensajes de salida hacia otros servidores a los que otros clientes se conectarán en su momento para descargarlos.

Otro protocolo, un tanto en desuso hoy día, es el de transferencia de archivos FTP (*File Transfer Protocol*). A través de esta aplicación es posible descargarse en la máquina local cualquier archivo de cualquier tipo que esté almacenado en una máquina remota. Para ello son necesarios un cliente y un servidor de FTP. El servidor está ejecutándose continuamente en la máquina remota en la que están los archivos que se quieren descargar, a la espera de que algún cliente le solicite alguno. Cuando esto ocurre, solicitará generalmente la identificación del cliente mediante un *login* y clave, si bien muchos admiten un usuario llamado *anonymous* que puede entrar sin clave. Una vez identificado el usuario, le dará acceso a una parte determinada del disco donde están almacenados los archivos, de modo que cada usuario sólo podrá ver aquellos a los que tenga permiso de acceso (el usuario *anonymous* solía tener acceso a una zona donde había *software* y documentos públicos). El cliente es el programa que el usuario ejecuta para conectarse con el servidor e identificarse ante él. Una vez conectados, a través del cliente el usuario podía enviar diferentes peticiones al servidor, relacionadas con el manejo de archivos, tales como ver un listado de ellos, cambiar de carpeta, borrar archivos, renombrarlos, transferirlos a la máquina local (*bajarlos*) o copiar archivos locales en la máquina remota (*subirlos*). Hoy día el protocolo FTP es poco usado porque la comunicación entre cliente y servidor no va cifrada, lo que significa que los datos que se transfieren pueden ser interceptados por alguien que tenga acceso físico a la red que los comunica (y eso incluye el *login* y contraseña). La versión cifrada (segura) del protocolo FTP se denomina FTPS, pero incluso ese se usa poco porque el protocolo HTTP permite hacer básicamente las mismas cosas, y no se requiere un cliente específico para ello, sino que el mismo navegador Web sirve.

La implementación del paradigma cliente-servidor puede realizarse con diferentes modelos de comunicación, como el paso de mensajes que vimos anteriormente, pero también la llamada a procedimientos remotos, la invocación de métodos remotos o los servicios web. Veremos estos modelos de comunicación a continuación.

La API de *sockets* sobre TCP implementa de forma natural este modelo, al existir *sockets* de dos tipos: el pasivo, que básicamente sólo sirve para permanecer a la espera de conexiones, y el *activo* que es aquél que realmente puede comunicar datos. Un servidor utiliza un `socket` pasivo para esperar las conexiones de los clientes, y crea otro activo cuando recibe una conexión, para comunicarse.

## 5.3 Llamadas a procedimientos remotos (RPC)

La llamada a una función o procedimiento en un lenguaje procedimental como C, Java o Python puede verse en cierta forma como un paso de mensajes. Considera la ejecución de una línea de código como la siguiente:

Java

```
resultado = operacion(a,b,c);
```

Esa llamada puede verse desde la óptica del paradigma cliente-servidor, si la consideramos en la

forma siguiente:

- El programa principal necesita de un "servicio" que no forma parte de su código, sino que está en una biblioteca externa, implementado en una función. El "servicio" consiste en realizar una cierta `operacion` con las cantidades `a`, `b` y `c`. Realiza una llamada que puede verse como una "petición".
- En la llamada se transmiten los parámetros `a`, `b` y `c` a la función. El paso de parámetros ocurre en realidad a través de la pila, pero puede verse como un paso de mensajes.
- El "cliente", que es el programa que ha hecho la llamada, queda a la espera de que el "servidor" (que es la función `operacion`) finalice y retorne el resultado.
- Cuando la función termina su ejecución devuelve un resultado, lo cual puede verse de nuevo como un paso de mensajes que constituye la "respuesta" al servicio solicitado.

Así pues, si una llamada a un procedimiento puede verse como caso de interacción cliente-servidor ¿por qué no al contrario? Es decir, implementar una interacción cliente-servidor real, que tiene lugar entre máquinas separadas, usando sintaxis de llamada a función.

Esta es la poderosa idea detrás del concepto de RPC. Es una idea que permite desde el código de un cliente poder invocar a funciones que están implementadas en un servidor, usando la misma sintaxis que se usaría para una llamada local.

Naturalmente para que esto sea posible ha de haber un software intermedio (*middleware*) que se ocupe de la transmisión real de los parámetros y resultados a través de la red. Este problema es mucho más complejo de lo que parece a primera vista si se tiene en cuenta que no basta con transmitir la representación binaria de esos datos, ya que las máquinas en que se implementen cliente y servidor podrían tener diferente arquitectura y por tanto representar los mismos datos con diferentes patrones de bits. Por no mencionar qué ocurre si el cliente intenta transmitir un puntero al servidor. El puntero es una variable que apunta a una dirección de memoria dentro de la máquina del cliente, y por tanto no es útil enviarlo "tal cual" al servidor.

El *middleware* tiene que ocuparse no sólo de la correcta transmisión de la información entre cliente y servidor, sino también de determinar cómo hacer llegar la información a ese servidor. ¿Cuál es la IP del servidor? ¿en qué número de puerto está escuchando? ¿cómo indicarle qué función queremos que sea ejecutada en el servidor? ¿cómo saber qué funciones tiene implementadas?

De estos problemas y sus soluciones nos ocuparemos en el tema 2.

## 5.4 Invocación remota de métodos (RMI)

El concepto de RMI es exactamente el mismo que el de RPC, pero aplicado al campo de la programación orientada a objetos (OOP).

En la OOP las funciones no existen generalmente disociadas de objetos, sino en forma de métodos pertenecientes a una clase instanciada habitualmente en un objeto. No obstante puede seguirse la misma línea de razonamiento que en el apartado anterior y ver que una invocación de un método en un objeto también se puede contemplar como un caso de interacción cliente-servidor, siendo el cliente el programa que realiza la invocación y el servidor el propio objeto que ejecuta ese método.

Todos los problemas mencionados en relación a las RPC aparecen también en las RMI, junto con algún problema adicional relacionado con la programación orientada a objetos. No obstante los conceptos son básicamente los mismos.

## 5.5 Redes *Peer to peer* (P2P)

El término *peer to peer* se traduce como "de igual a igual" o "entre iguales". En este modelo no existe la distinción de roles cliente/servidor. En lugar de ello, todos los procesos que participan en la comunicación son iguales entre sí. Puede tratarse de hecho del mismo código ejecutándose en diferentes máquinas.

El hecho de que no exista distinción entre procesos servidores y procesos cliente, significa que cualquiera de ellos puede iniciar una comunicación hacia otro cualquiera de los participantes en cualquier momento. Y esto implica también que todos ellos deben estar preparados para recibir un mensaje de cualquiera de los restantes, en cualquier momento. En cierta forma, todos los procesos son servidores ya que todos han de estar a la escucha de mensajes que provengan de los restantes, y a la vez todos son clientes ya que cualquiera puede iniciar un mensaje hacia otro.

Este modelo de interacción facilita la implementación de algoritmos descentralizados, ya que por definición no hay "centro" en una red P2P, al ser todos los procesos iguales entre sí.

**Por ejemplo**, imagina la situación en la que se tiene un archivo muy voluminoso (por ejemplo 4Gb), que está físicamente almacenado en una máquina A, y se desea hacer muchas copias del mismo en otras muchas máquinas conectadas a la máquina A y entre sí por una red. Podría tratarse por ejemplo de una imagen `iso` de instalación de algún programa, o una actualización de un sistema operativo que hay que aplicar a todas esas máquinas.

El enfoque tradicional cliente-servidor consistiría en instalar un servidor en la máquina que tiene el archivo, y un cliente en todas las demás máquinas. Por ejemplo, podría usarse el protocolo FTP de transferencia de archivos, o el HTTP.

Esta solución causará un importante cuello de botella en el servidor, ya que todos los clientes están intentando descargar la información a la vez del mismo lugar. El ancho de banda de salida del servidor se saturará, y quizás también su CPU o el ancho de banda con el disco duro. Como consecuencia la transferencia será muy lenta.

En una solución *peer to peer* se instalaría software P2P en todas las máquinas. Todas llevarían el mismo software. Ya que inicialmente sólo una de ellas tiene el archivo que se quiere compartir, ésta actuará inicialmente como servidor. Algunas de las otras máquinas se conectarán a ella y solicitarán un trozo del archivo, elegido al azar. Tan pronto como una máquina cliente dispone de ese trozo, se convierte a su vez en servidor para los demás clientes.

En poco tiempo no se tendrá un servidor, sino muchos. Cada cliente se conectará a varios de estos servidores a la vez, para bajarse de cada uno de ellos un trozo diferente del archivo. El ancho de banda de la red se aprovecha al máximo de este modo, sin crearse cuellos de botella en ninguna de las máquinas. Se logra de este modo que todas las máquinas consigan su copia del archivo en el menor tiempo posible.

El diseño de este tipo de sistemas plantea importantes retos. Al no existir un servidor central que pueda almacenar información sobre la configuración del sistema, esta información ha de estar distribuida entre los diferentes nodos que lo componen. Deben diseñarse estructuras de datos y algoritmos que permitan trabajar con información distribuida, descubrir la adición o desaparición de nodos en el sistema, adaptarse en tiempo real a esa topología, etc. Hay también muchos aspectos relacionados con la seguridad, la privacidad, etc.

## 5.6 Agentes móviles

Un agente móvil es un programa u objeto (si hablamos de programación orientada a objetos) *transportable*. Es decir, que está autocontenido (código y datos) de forma que pueda pausar su ejecución en una máquina, copiarse a través de algún medio hasta otra máquina y reanudar la ejecución en su destino.

Ya vemos que para que esto sea posible, o bien ambas máquinas tienen la misma arquitectura y sistema operativo, o bien el agente está programado en un lenguaje independiente de la arquitectura (como Java o Python).

La idea es que el agente se lanza en una máquina, y es él mismo quien decide cuándo “moverse” a otra, y tiene capacidad de hacerlo de forma autónoma. Es decir, el agente tiene código para conectar con otra máquina, copiarse allí, terminar su ejecución en la máquina local y reanudarla en la máquina remota. Parece evidente que esto no lo puede hacer por sí solo, sino que requerirá de algún *middleware* que se ocupe de “recibirle” en la máquina destino y de lanzar su ejecución.

En cada “salto” el agente permanece en ejecución un tiempo en la máquina que lo recibe, durante el cual aprovecha los recursos de esa máquina para avanzar en su tarea. Se entiende que hará uso de recursos específicos de esa máquina, que no puede encontrar en otro lugar, y que cuando ya tiene lo que buscaba, realiza el salto a otra para continuar.

Es importante señalar que los agentes no necesitan intercambiar mensajes entre sí. Cada agente es

código autocontenido. Esto reduce el tráfico de red, y posibilita que los agentes puedan seguir funcionando y haciendo su trabajo incluso en entornos donde la red funciona de forma intermitente o errática.

Un posible **ejemplo** de la utilidad de este paradigma podría ser el siguiente: se necesita una información que está en una base de datos remota. La búsqueda a realizar es lo bastante complicada como para que no se resuelva con una simple consulta SQL. Necesita primero obtener información de la base de datos, y en función de la información obtenida, realizar más consultas, agregar los resultados, etc. En definitiva, necesita escribir un programa que dirija todo el proceso. Además el usuario iniciará su búsqueda desde su teléfono móvil.

En un paradigma cliente-servidor, escribiríamos el código del cliente para ejecutarse en el teléfono, y ese código usaría la red de datos del teléfono para conectar con el servidor. Debido a que tiene que hacer varias conexiones y en cada una de ellas puede recibir mucha información, estaría haciendo un elevado consumo de datos, y de batería.

En un paradigma de agentes móviles, escribiríamos el código del agente. El usuario lanzaría el agente desde su teléfono para instalarlo en el servidor donde está la base de datos. El agente, una vez en su destino, realizaría todas las consultas locales y una vez tenga la respuesta se la comunicaría al teléfono del usuario. Como vemos, mientras el agente trabaja, el teléfono no usa CPU ni datos, por lo que es una solución más rápida y económica (aunque para funcionar se requiere obviamente la colaboración del servidor que debería contar con una infraestructura adecuada para recibir y ejecutar agentes).

Se comprende que aunque este modelo es útil, plantea importantes problemas en aspectos de seguridad y fiabilidad. ¿Cómo gestionar los permisos de qué agentes pueden ejecutarse y qué cosas pueden hacer? ¿Y si el código que trae el agente contiene bugs que comprometen el funcionamiento de la máquina que lo recibe? ¿Y si contiene código malicioso?

## 5.7 Servicios Web

El *World Wide Web* (WWW) o simplemente "*la web*", nació como un medio para intercambiar documentos (mayormente texto con formato HTML, y con posibilidad de soportar imágenes). Los servidores Web eran básicamente servidores de documentos que recibían peticiones bajo el protocolo HTTP, y respondían enviando el documento solicitado.

A través de la interfaz CGI, la Web permitía también que el documento solicitado pudiera no existir realmente en ninguna parte del disco duro del servidor, sino que ante la petición adecuada el servidor ejecutaba un programa, y era la salida de ese programa la que constituía la respuesta que se enviaba al cliente.

Esto hacía posible implementar funcionalidades que se ejecutaban en el servidor a partir de una petición iniciada desde un navegador Web. Por ejemplo, el usuario podía rellenar un formulario HTML y

pulsar un botón para enviarlo, ante lo cual el navegador construía un mensaje `POST` (parte del protocolo HTTP) que incluía como parámetros los valores de los campos extraídos del formulario, y enviaba esa petición al servidor. Éste, al recibirla, podía ejecutar un *CGI-script* que era un programa alojado en el servidor (que podía estar escrito en cualquier lenguaje de programación, si bien típicamente se trataba de código en el lenguaje del *shell* o en Perl), y ese programa podía acceder a los parámetros que el navegador había enviado.

A partir de esa petición y de los parámetros suministrados, el *script* invocado podía llevar a cabo algún tipo de servicio (búsqueda en una base de datos, almacenamiento de información acerca del usuario, compra de un producto, transacción bancaria, etc..) y construir una respuesta en formato HTML como resultado.

La respuesta era enviada al navegador Web, quien la mostraba al usuario.

Así pues vemos que el Web ya contaba con la infraestructura para realizar una especie de invocación remota. Desde el cliente, se ejecutaba código en el servidor. El problema era que toda esta comunicación estaba orientada al "consumo humano", es decir, el resultado generado desde el servidor era código HTML, diseñado (a veces pobremente) para ser mostrado en un navegador y ser leído por una persona.

Enseguida se hace evidente que, si se quieren automatizar tareas sin requerir la intervención del usuario, se puede escribir un programa que actúe como el navegador web, enviando al servidor los datos que necesita el servicio, (tomados quizás de variables del programa o de una base de datos local, en lugar de preguntárselos al usuario a través de un formulario). Después, se recibe la respuesta del servidor, que estará en lenguaje HTML y será necesario *parsear* para extraer de entre todos los *tags* la información relevante y almacenarla en otras variables o archivos locales.

El problema es que lo anterior no deja de ser un *hack*, es decir una solución ingeniosa para usar algo de una forma que no era para la que había sido pensada. Además, la labor de parsear el HTML era compleja, sobre todo teniendo en cuenta que la mayoría de los servidores enviaban HTML *mal formado* (tags sin cerrar, mal anidamiento, etc.)

Los **servicios Web** son la formalización y estandarización de la idea anterior.

Se trata de una implementación más del concepto de "llamada a procedimiento remoto", pero usando los protocolos y estándares del Web. En concreto:

- El cliente realiza las invocaciones mediante el protocolo HTTP (comandos `GET`, `POST`, `PUT` y raramente `DELETE`).
- El servicio invocado se identifica por su *URL*.
- El servidor contiene un *middleware* que transforma esa petición en una invocación de una función (o de un método de un objeto residente en el servidor), ocupándose de traducir apropiadamente los parámetros y resultados.

- El lenguaje XML es utilizado intensivamente para:
  - Describir los servicios ofertados, y su especificación (número y tipo de los parámetros, los resultados, etc.)
  - Codificar los parámetros que envía el cliente
  - Codificar la respuesta del servidor

  La ventaja de XML es que es un lenguaje formal, estricto y para el que existen ya un buen número de *parsers* escritos, que hacen sencilla la extracción de la información y la conversión al formato de representación interno del cliente o el servidor.

Gracias a lo anterior, es posible *exponer* muchos de los servicios que un servidor ofertaba a sus usuarios humanos, de modo que puedan ser consumidos también por usuarios "no humanos", es decir, otros programas.

Una ventaja importante de este enfoque es que, al ir toda la comunicación basada en el protocolo HTTP que es el mismo que usan los navegadores Web, la comunicación puede pasar a través de Proxies y cortafuegos sin encontrarse con los problemas asociados a otras tecnologías. La comunicación a través del puerto 80 (el usado por HTTP), siempre está abierta, pues de lo contrario los usuarios no podrían navegar por la Web.

Ejemplos de implementaciones de esta idea son SOAP (si bien SOAP está diseñado de modo que admita más métodos de transporte aparte del HTTP), y más recientemente un enfoque más ligero denominado RESTful (o REST) que usa JSON en lugar de XML para codificar los parámetros y las respuestas, y se basa fuertemente en el protocolo HTTP (verbos y URLs) para hacer más explícito y claro el cometido de las diferentes interacciones. REST ha reemplazado casi por completo a SOAP debido a la complejidad y sobrecarga que imponía éste.

REST se ha convertido en una forma muy estándar de ofrecer servicios que puedan ser invocados desde otros clientes, usando HTTP para ello. Esto permite diseñar *Arquitecturas Orientadas a Sercvicios* (o microservicios) en las que cada componente software es un proceso independiente, que se puede estar ejecutando en una máquina separada, y que interactúan entre sí a través de interfaces REST. Es decir, una forma de crear sistemas distribuidos.

1. Famoso también por ser el autor del software *ghostscript*, un intérprete de lenguaje PostScript que permitió visualizar este tipo de documentos, muy comunes antes de la llegada del PDF, en un ordenador de sobremesa, y de imprimirlos en una impresora convencional, sin necesidad de adquirir un costoso *display* o impresora con soporte PostScript. El software todavía se usa a día de hoy para convertir documentos `ps` a otros formatos.↩