UCAB UNIVERSIDAD CATÓLICA ANDRÉS BELLO

SISTEMAS DISTRIBUIDOS, ESCALABILIDAD Y DISTRIBUCIÓN DE LOS DATOS

Sistemas distribuidos, escalabilidad y distribución de los datos. por Mercy Ospina- UCAB, está bajo una Licencia Creative Commons Atribución-NoComercial-CompartirIgual 4.0 Internacional.

OBJETIVO

Comprender qué es un sistema distribuido, sus niveles de transparencia, las arquitecturas y la escalabilidad que presentan y cómo se pueden distribuir los datos en esta clase de sistema.

TABLA DE CONTENIDO

> Introducción

01 Sistemas distribuidos, niveles de transparencia.

02 Arquitecturas de los sistemas distribuidos.

03 Escalabilidad y distribución de los datos.

> Cierre

> Referencias

Las propiedades de volumen y velocidad de las versus de *big data* son posibles gracias a **la evolución de los sistemas distribuidos**. Debido a esto, una construcción de una solución de *big data* debe pasar por la comprensión de lo que es un sistema distribuido, sus niveles de transparencia, las diferentes arquitecturas y características, la escalabilidad y las técnicas de distribución de datos.

Hasta la década de los 80 las computadoras fueron grandes y caras; sin embargo, a mitad de esta década, dos avances de la tecnología comenzaron a cambiar esta situación: el primero de ellos fue el desarrollo de **poderosos microprocesadores**. Inicialmente, los microprocesadores eran máquinas de 8 *bits*, pero pronto se hicieron comunes las CPU de 16, 32 y 64 *bits*, y la cantidad de mejoras en este particular ha sido verdaderamente impresionante.

El segundo desarrollo importante fue la invención de las redes de **computadoras de alta velocidad**. Las redes de área local, o LAN *(local-area networks)*, permiten la interconexión de cientos de máquinas localizadas dentro de un mismo edificio, de tal manera que es posible transferir grandes volúmenes de datos entre máquinas, a velocidades que van de los 100 millones a los 10 millones de millones de *bits* por segundo. Las redes de área amplia, o WAN (*wide area networks*), facilitan la interconexión de millones de máquinas ubicadas alrededor del mundo a velocidades que van desde los 64 Kbps (*kilobits* por segundo) hasta *gigabits* por segundo.

El resultado de estas tecnologías es que ahora no solamente es factible sino también fácil, cantidades poner a trabajar sistemas de cómputo compuestos por grandes cantidades de computadoras interconectadas mediante una red de alta velocidad.

Por lo general, a estos sistemas se les conoce como redes de computadoras o sistemas distribuidos, al contrario de los sistemas centralizados (o sistemas de un solo procesador) que usualmente constan de una sola computadora, sus periféricos y quizás algunas terminales remotas (Tanenbaum y Steen, 2008).

1. Definición de sistemas distribuidos:

En la **literatura** especializada existen varias definiciones de sistemas distribuidos. Entre ellas:

- "Colección de computadoras independientes que dan al usuario la impresión de constituir un único sistema coherente" (Tanenbaum y Steen, 2008.)

- "Conjunto de computadoras independientes, interconectadas a través de una red y que son capaces de colaborar a fin de realizar una tarea" (Liu, 2004.)

Estas definiciones comprenden aspectos importantes. En primer lugar, tenemos que un sistema distribuido consta de **componentes** (es decir, computadoras) autónomos. El segundo aspecto es que los usuarios (personas o programas) creen que realmente **interactúan** con un sistema único. Esto significa que, de una manera o de otra, los componentes autónomos necesitan colaborar entre sí. La forma de establecer la colaboración es la base de los sistemas distribuidos.

Algunas de las características importantes **de los sistemas distribuidos** son:

- Las **diferencias** entre las distintas computadoras y la manera en que se comunican entre sí quedan ocultas para el usuario y lo mismo sucede con la organización interna de un sistema distribuido.

- Los usuarios y las aplicaciones pueden **interactuar** con un sistema distribuido de manera consistente y uniforme, sin importar dónde y cuándo tenga lugar.

- En principio, deben ser fáciles de **expandir** o escalar; los usuarios y las aplicaciones no deben notar que las partes son reparadas o que se agregan nuevas secciones para servir a más usuarios o aplicaciones.

El **principal objetivo** de un sistema distribuido es facilitar a los usuarios (y a las aplicaciones) el acceso a los recursos remotos y compartirlos de manera controlada y eficiente, lo que permite abaratar costos, colaborar e intercambiar información. Sin embargo, mientras la conectividad y el intercambio aumentan, la seguridad se vuelve cada vez más importante, ya que es necesario entonces proporcionar protección en contra del espionaje o de la intrusión en las comunicaciones.

2. Transparencia en la distribución:

Un objetivo importante de un sistema distribuido es disfrazar el hecho de que sus procesos y recursos están físicamente distribuidos a través de **múltiples computadoras**. Decimos que un sistema distribuido es transparente si es capaz de presentarse ante los usuarios y las aplicaciones como si se tratara de una sola computadora. Primero echemos un vistazo a los tipos de transparencia que existen en los sistemas distribuidos. Después, enfrentaremos la pregunta general con respecto a si la transparencia es siempre requerida.

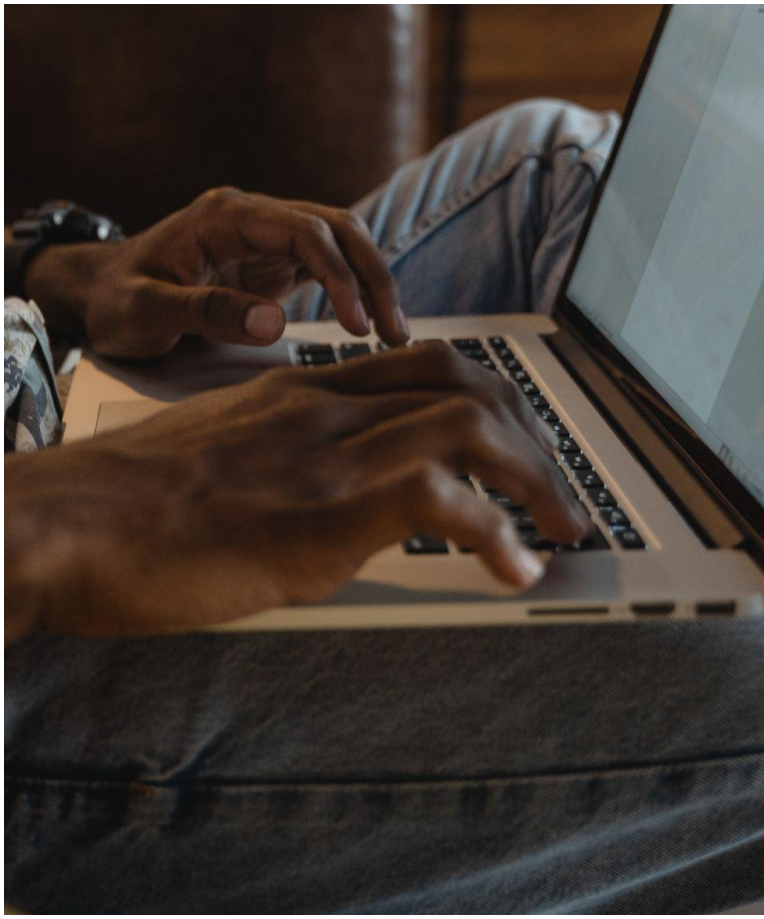
Tipos de transparencia: es posible aplicar el concepto de transparencia a distintos aspectos de un sistema distribuido. La tabla 1 muestra una relación de los tipos más importantes:

Transparencia	Descripción
Acceso	Oculta diferencias en la representación de los datos y la forma en que un recurso accede a los datos
Ubicación	Oculta la localización de un recurso
Migración	Oculta el que un recurso pudiera moverse a otra ubicación
Reubicación	Oculta el que un recurso pudiera moverse a otra ubicación mientras está en uso
Replicación	Oculta el número de copias de un recurso
Concurrencia	Oculta que un recurso puede ser compartido por varios usuarios que compiten por él
Falla	Oculta la falla y recuperación de un recurso

Fuente: Tabla 1. Distintas formas de transparencia aplicables en un sistema distribuido. Extraída de ISO (1995)

La transparencia de acceso se encarga de esconder las diferencias en la representación de los datos y la manera en que el usuario accede a dichos recursos. Es decir, las diferencias en la convención de nombres, así como la manera en que podemos manipular los archivos, deberán quedar ocultas para los usuarios y las aplicaciones. La transparencia de ubicación, por su parte, se refiere al hecho de que los usuarios no pueden determinar en qué ubicación física se localiza el recurso en el sistema.

Los sistemas distribuidos en los cuales es posible reubicar los recursos sin afectar la manera en que podemos acceder a dichos recursos proporcionan transparencia de migración. Es incluso **más importante** la situación en la cual podemos reubicar los recursos mientras disponemos de ellos sin que el usuario o la aplicación lo noten. En tales casos, decimos que el sistema permite una transparencia de reubicación. Un ejemplo de transparencia de reubicación es cuando los usuarios móviles pueden continuar usando sus computadoras portátiles inalámbricas mientras se mueven de un lugar a otro sin desconectarse (temporalmente).



Por otro lado, la replicación juega un papel muy importante en los sistemas distribuidos y en ***big data***. Por ejemplo, podemos replicar los recursos para incrementar la disponibilidad o para mejorar el rendimiento. La transparencia de replicación tiene que ver con encubrir que existen distintas copias del recurso.

Si se requieren recursos de manera compartida (por ejemplo, la misma tabla de una base de datos), es importante que cada usuario no advierta que otros están haciendo uso del mismo recurso; a eso se le llama transparencia de concurrencia. Finalmente, la transparencia de fallas consiste en que los usuarios del sistema no perciban ni cuando un recurso ha dejado de funcionar, ni cuando el sistema se recupera después del error.

Nivel de transparencia: aunque la transparencia de distribución generalmente es considerada preferible para cualquier sistema distribuido, existen situaciones en las que tratar de ocultar por completo todos los aspectos de distribución a los usuarios no es una buena idea. Existe cierto intercambio entre un alto grado de transparencia y el rendimiento del sistema. Por ejemplo, muchas aplicaciones de internet tratan repetidamente de contactar a un servidor antes de darse por vencidas. En consecuencia, intentar enmascarar la falla de un servidor transitorio antes de buscar el contacto con otro servidor pudiera volver más lento a todo el sistema. En tal caso, podría ser preferible desistir antes o al menos permitir que el usuario intente hacer contacto.

También se presentan situaciones donde no resulta muy evidente que ocultar la distribución sea buena idea. Conforme los sistemas distribuidos se expanden hacia los dispositivos que los usuarios traen consigo (móviles y ubicuos), donde la simple noción de la ubicación y de la idea de contexto se vuelve cada vez más **importante**, podría ser más efectivo exponer la distribución, en lugar de tratar de ocultarla.

Otros argumentos en contra de la transparencia de distribución son, por ejemplo, que reorganizarla por completo es simplemente imposible, por lo cual se hace necesario preguntarse si es inteligente pretender que se puede conseguir. En otras palabras, sería mucho mejor hacer una distribución explícita de modo que nunca engañemos al desarrollador de la aplicación ni al usuario. El resultado será que los usuarios comprenderán mucho mejor el (a veces inesperado) comportamiento de un sistema distribuido y estarán entonces mucho mejor preparados para interactuar con ese comportamiento.

Por lo general, los sistemas distribuidos son **complejas** piezas de *software* cuyos componentes se encuentran, por definición, dispersos en diversas máquinas.

Para dominar esta complejidad, resulta crucial que los sistemas se encuentren **organizados** adecuadamente.

Existen diferentes formas de visualizar la organización de un sistema distribuido, pero un modo evidente es saber diferenciar la organización lógica de la colección de componentes del *software* de la organización física real.

La organización de los sistemas distribuidos trata básicamente sobre los componentes de *software* que constituyen el sistema. Estas arquitecturas de *software* nos dicen cómo se organizarán los componentes de *software* y cómo deben interactuar.

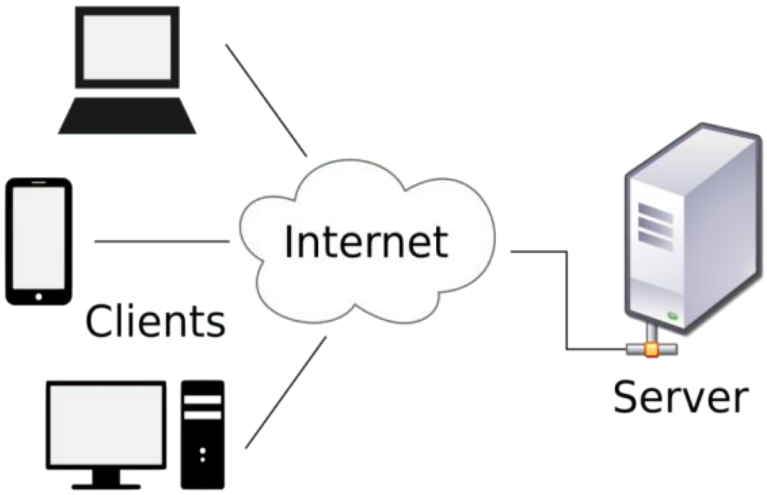
Para este curso, la arquitectura se define de acuerdo a cómo los componentes del sistema distribuido interactúan entre sí y en cómo estos se configuran juntos en el sistema.

1. Arquitecturas centralizadas:

Estas arquitecturas están basadas en términos de clientes que **requieren** servicios de los servidores.

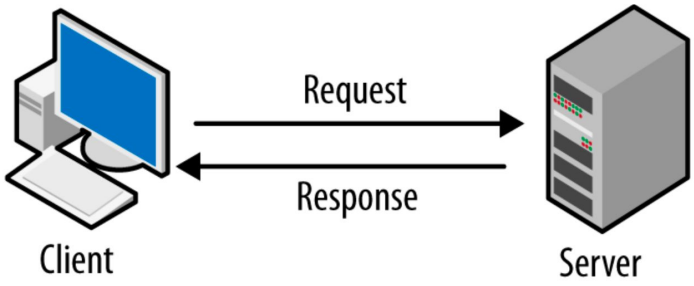
En el modelo básico cliente-servidor, los procesos de un sistema distribuido se dividen en dos grupos (probablemente traslapados):

- **Un servidor** es un proceso que implementa un servicio específico, por ejemplo, un servicio de sistema de archivos o un servicio de base de datos.
- **Un cliente** es un proceso que solicita un servicio a un servidor, enviándole una petición y esperando posteriormente la respuesta.



Fuente: Arquitectura cliente-servidor. Extraída de por Nuñez (2020).

Una relación cliente-servidor corresponde al patrón de solicitud-respuesta y debe adherirse al procedimiento de comunicaciones común que define el lenguaje, las reglas o los patrones de diálogo utilizados. Por ejemplo, el conjunto de protocolos TCP o IP.



Fuente: Interacción solicitud–respuesta de la arquitectura cliente-servidor. Extraída de Darvish (2020).

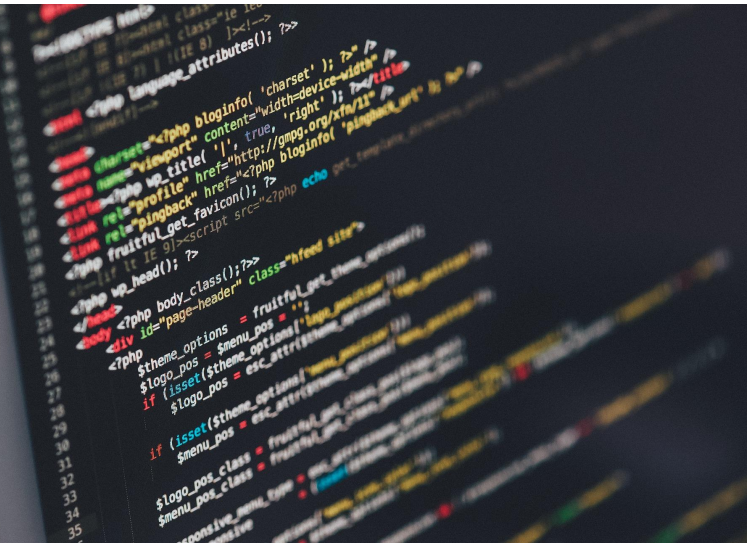
Aplicación de capas: a través del tiempo, el modelo cliente-servidor ha estado sujeto a muchos debates y controversias. Una de las principales cuestiones en discusión ha sido cómo establecer una diferencia clara entre un cliente y un servidor. No sorprende que con frecuencia, esta no exista. Por ejemplo, un servidor para una base de datos distribuida puede actuar continuamente como un cliente, ya que reenvía solicitudes a diferentes servidores de archivos responsables de implementar las tablas de la base de datos. En tal caso, el propio servidor de la base de datos no hace más que procesar consultas.

Sin embargo, si consideramos que muchas aplicaciones cliente-servidor están enfocadas en dar a los usuarios acceso a las bases de datos, mucha gente ha **defendido** una diferencia entre los siguientes tres niveles, siguiendo básicamente el estilo arquitectónico en capas que explicamos antes:

- El nivel de interfaz de usuario.
- El nivel de procesamiento o *middleware*.
- El nivel de datos (servidor de base de datos).

Fuente: Arquitectura de 3 capas. Extraída de Singh (2017).

El nivel de interfaz de usuario comprende todo lo que se necesita para la **interacción directa** con el usuario, tal como la administración visual; el nivel de procesamiento contiene por lo general las aplicaciones; y el nivel de datos administra los datos reales sobre los que se está actuando.



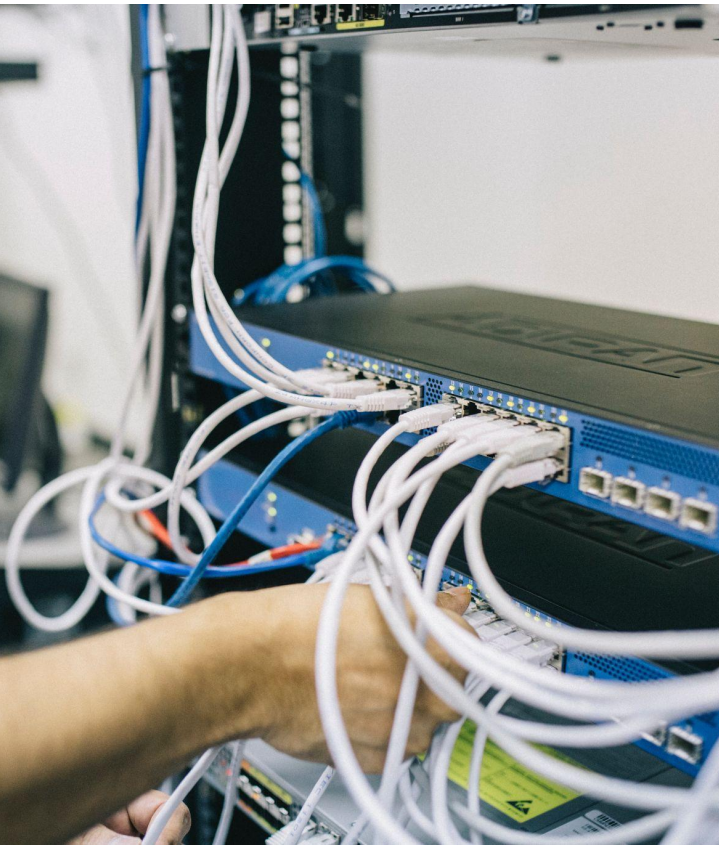
Los clientes comúnmente **implementan** el nivel de interfaz de usuario. Este nivel consiste en los programas que permiten a los usuarios finales interactuar con aplicaciones.

Muchas aplicaciones cliente-servidor pueden construirse a partir de **tres partes** diferentes: una que maneja la interacción con el usuario, otra que opera sobre una base de datos o sistema de archivos, y una parte intermedia que generalmente contiene la funcionalidad principal de una aplicación.

Clúster de servidores: como ya hemos dicho, un servidor es un proceso que aplica un servicio específico para un conjunto de clientes. Existen distintas maneras de organizar servidores.

En el cómputo en clúster, el *hardware* subyacente consta de una colección de estaciones de trabajo similares, o computadoras personales independientes (cada uno con su propia memoria local), por lo general servidores, llamadas nodos, conectadas cercanamente por medio de una red de área local de alta velocidad, con el objetivo de realizar una tarea específica (Tanenbaum y Steen, 2008).

Acá uno o más nodos maestros se encargan de **distribuir** las tareas, balanceando las cargas entre los demás nodos (esclavos).



Los clústeres son usualmente empleados para mejorar el rendimiento o la disponibilidad por encima de la que es provista por un solo computador, típicamente siendo esta una opción más **económica** que el uso de computadores individuales de rapidez y disponibilidad comparables; de allí su aplicación en *big data*, donde la capacidad de cómputo y almacenamiento sobrepasa con creces la de un único computador.

De un clúster se espera que presente combinaciones de los siguientes servicios:

- **Alto rendimiento**: poder ejecutar tareas que requieren de gran capacidad computacional, grandes cantidades de memoria, o ambos a la vez. El llevar a cabo estas tareas puede comprometer los recursos del clúster por largos períodos de tiempo.

- **Alta disponibilidad**: brindar la máxima disponibilidad de los servicios que ofrecen a través de la confiabilidad y tolerancia a fallos.

- **Balanceo de carga de trabajo**: distribuir las tareas del clúster de la forma más equitativa posible entre los esclavos.

- **Escalabilidad**: capacidad de aumentar el poder de cómputo de manera dinámica, agregando nodos al clúster.

La tecnología clúster permite a las organizaciones incrementar su capacidad de procesamiento usando tecnología estándar tanto en componentes de *hardware* como de *software*, que pueden adquirirse a un costo relativamente bajo.

2. Arquitecturas descentralizadas:

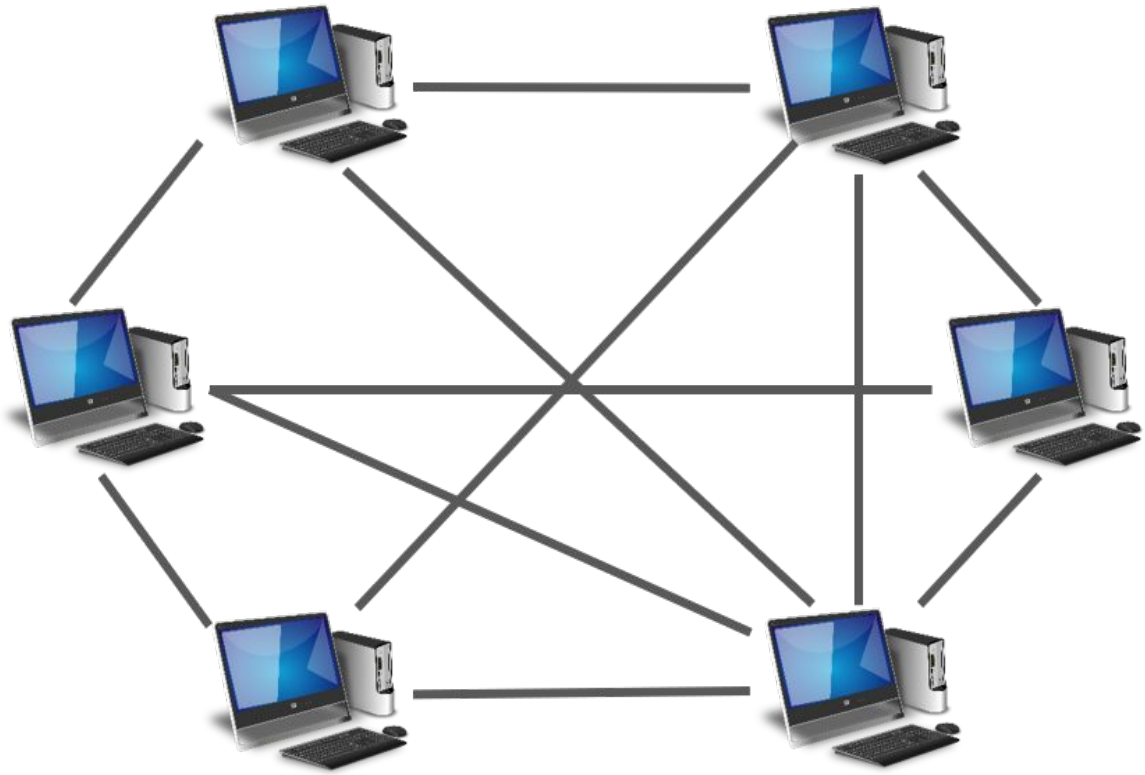
En esta sección veremos un tipo de arquitectura llamada "**sistema punto a punto**" (P2P) o "de pares". Desde una perspectiva de alto nivel, todos los procesos que constituyen un sistema de punto a punto son iguales. Como consecuencia, gran parte de la interacción ocurrida entre los procesos es simétrica: cada proceso actuará como cliente y servidor al mismo tiempo (Tanenbaum y Steen, 2008).

En esta arquitectura **no es necesario** un servidor central que administre la red (aunque puede existir), sino que todos los nodos de la red pueden comunicarse entre sí (Blancarte, 2021).

La arquitectura P2P es como si uniéramos al cliente y al servidor en una sola aplicación, **facilitando** la conexión a otras computadoras de la red para consumir los recursos expuestos por los otros nodos de la red. Pero al mismo tiempo, esta arquitectura funciona como un servidor, lo que permite que otros nodos se conecten a nuestro *software* para leer los recursos que nosotros exponemos.

Si bien P2P utiliza los mismos conceptos que una arquitectura cliente-servidor para conectar a todos los nodos de la red, la verdad es que persiguen la **solución de problemas diferentes**. Por una parte, la arquitectura cliente-servidor solo tiene un servidor, el cual centraliza los datos, recursos, seguridad, lógica de negocio, etc. De tal forma que, si el servidor se cae, se cae todo el sistema. Por otra parte, la arquitectura P2P busca que no exista un servidor central, sino que cada computadora dentro de la red funcione como un cliente y un servidor al mismo tiempo, de modo que, entre más usuarios conectados a la red, más servidores se unen también a la red.

En una arquitectura cliente-servidor, entre más usuarios conectados existan, más carga se acumula sobre el servidor; mientras que en una arquitectura P2P, entre más computadoras conectadas existan en la red, más poder de procesamiento se agrega, lo que la vuelve una red con una increíble capacidad de escalamiento.



Fuente: Arquitectura P2P. Extraída de Blancarte (2021).

Ventajas:

- **Alta escalabilidad:** las redes P2P permiten escalar fácilmente, pues a mayor número de nodos, más recursos hay disponibles y la carga de trabajo se divide entre todos los participantes.

- **Tolerancia a fallas:** tener los recursos distribuidos por varios nodos permite que los recursos estén disponibles sin importar si algunos de los nodos se caen.

- **Descentralización:** los sistemas P2P puros tienen una autonomía completa, por lo que pueden funcionar sin un servidor central que organice toda la red.

- **Privacidad:** debido al sistema de búsqueda por inundación o tablas hash, es posible tener un alto nivel de privacidad, pues no es tan fácil saber qué nodo fue el que realizó la petición inicial.

- **Equilibrio de carga:** en una arquitectura P2P, todos los nodos de la red tienen el mismo rol y responsabilidades, por lo que toda la carga de trabajo se equilibra entre ellos.

Desventajas:

- **Alta complejidad:** las arquitecturas P2P tienen una complejidad muy alta para desarrollarse, ya que tenemos que crear aplicaciones que funcionen como cliente y servidor, al mismo tiempo que tenemos que garantizar que las búsquedas de los recursos sean eficientes.

- **Control:** a menos que tengamos un servidor central para administrar los recursos que se buscan y los usuarios que se conectan a la red, es muy fácil perder el control sobre el contenido que se puede compartir en la red.

- **Seguridad:** esta es una de las características menos implementadas en este tipo de arquitecturas para evitar la intercepción de las comunicaciones, nodos maliciosos, contenido falso o adulterado, o la difusión de virus o programas maliciosos.

- **Tráfico:** en redes no optimizadas, como en las no estructuradas, podemos saturar la red con una gran cantidad de tráfico para propagar las peticiones a los nodos adyacentes.

Ya hemos nombrado este término anteriormente como una característica importante de los sistemas distribuidos. Según Mora Pérez (2012), la escalabilidad define la manera en que una plataforma de IT puede **crecer** para aumentar su capacidad de servicio y corresponde directamente con la arquitectura de la plataforma, ya que esta delimita la relación entre los componentes de dicha plataforma.

El análisis de la escalabilidad es el resultado del estudio de la escalabilidad de cada componente y de sus relaciones. Podemos estudiar la escalabilidad en:

- ***El software***: debido a que es el encargado con los datos y su escalabilidad está directamente relacionada a su capacidad de gestionar peticiones.
- ***El hardware***: ya que por su naturaleza dispone de un límite de capacidad para procesar los datos.

Existen dos formas en las cuales un componente puede escalar:

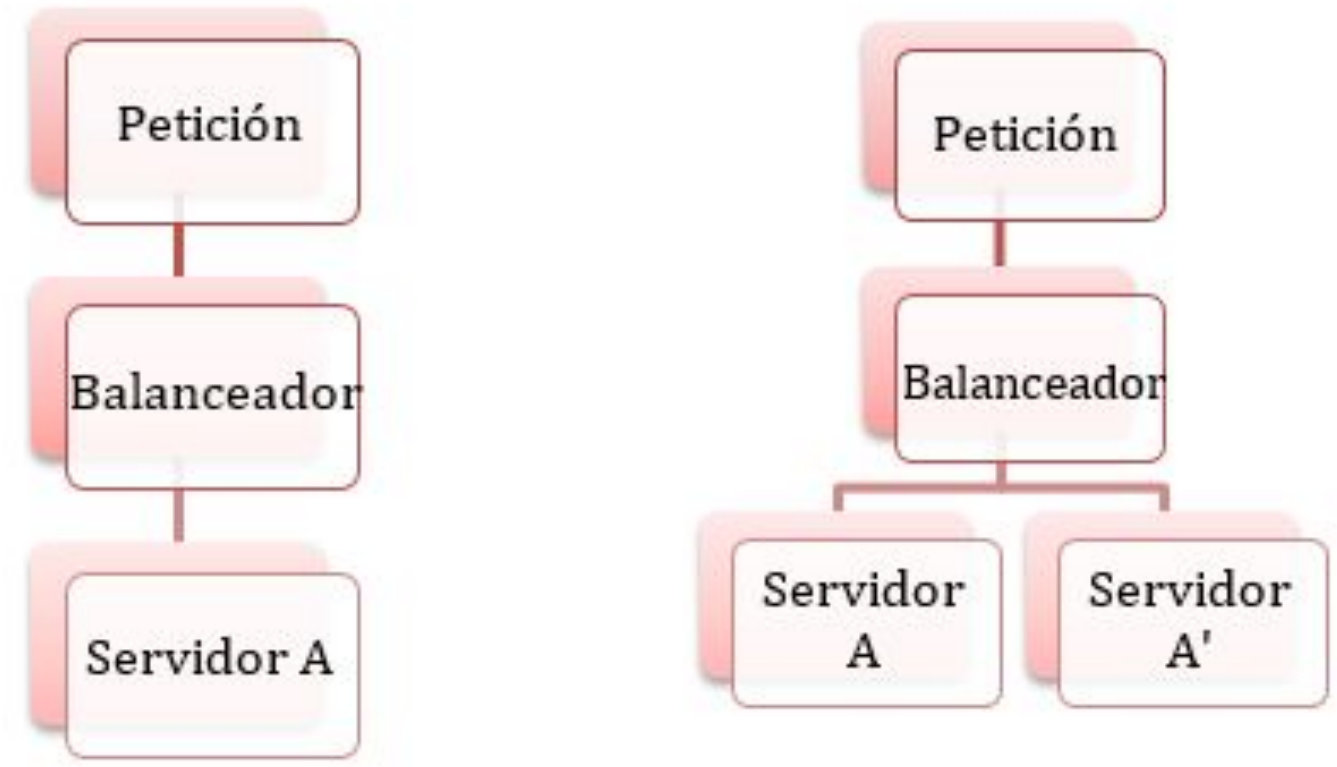
- **Vertical:** es la cualidad de aumentar la capacidad de sus recursos o componentes internos para incrementar su rendimiento. Ejemplo: aumentar la memoria o el disco duro de una PC.

- **Horizontal:** es la cualidad que tiene un componente de cooperar con componentes de su misma naturaleza y de esta manera incrementar el rendimiento de la tarea que se está realizando. Ejemplo: sistemas distribuidos, como un clúster de servidores.

La escalabilidad no es una propiedad que podamos aplicar de forma ilimitada, dado que existirán unos límites bien marcados, ya sea por limitaciones de capacidad de los componentes o por degradación de la capacidad del conjunto. Por ejemplo, en un sistema distribuido la limitación está principalmente en la capacidad de la red de comunicación, como en una autopista donde la velocidad del tráfico disminuye al haber más automóviles.

1. Balanceo de cargas:

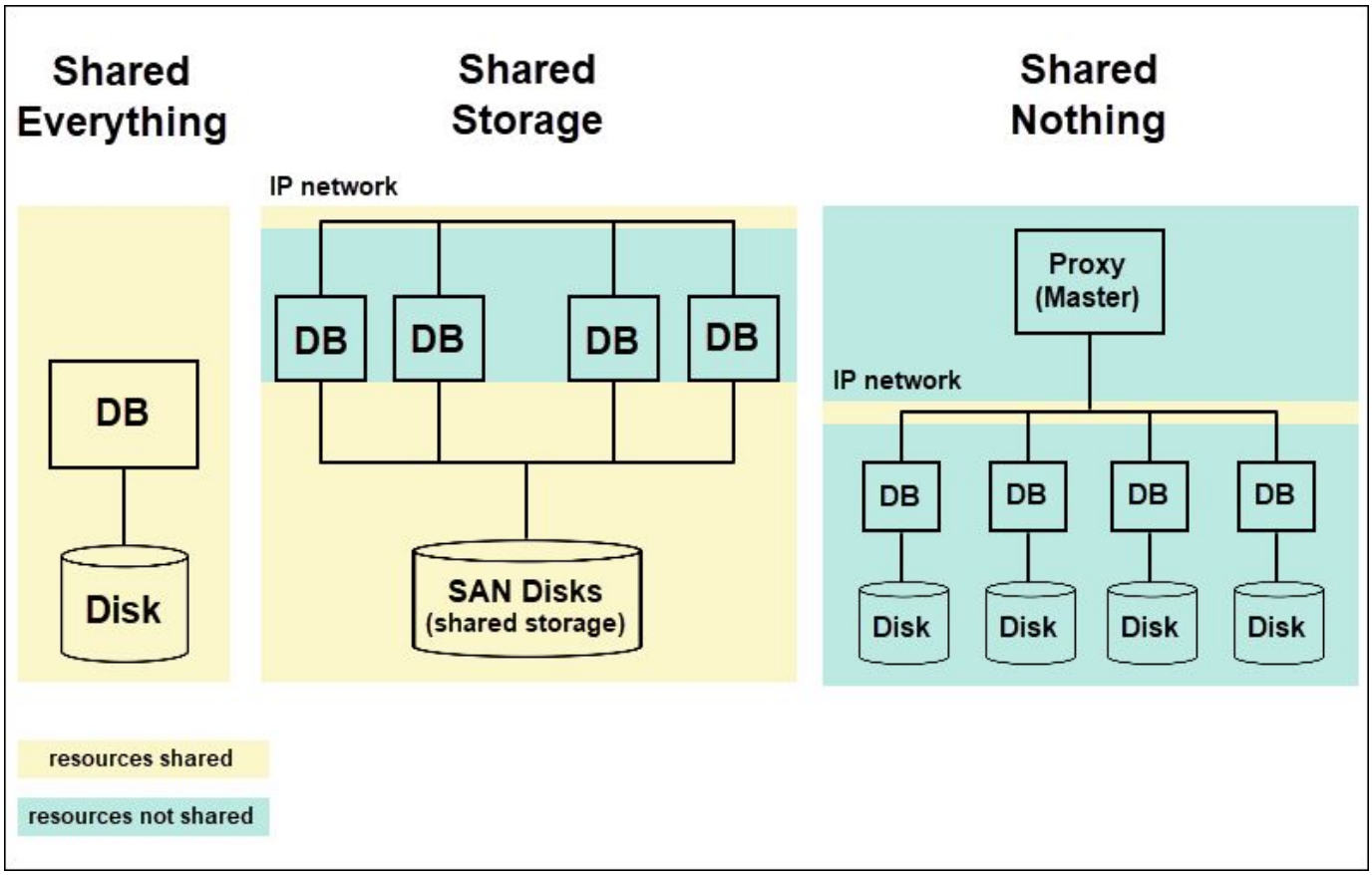
En el caso del escalamiento horizontal, existe el condicionante de que cada componente puede colaborar con los demás, por lo que se basa en poder **distribuir el trabajo** entre más componentes. Normalmente, esta función de distribución la realiza un elemento especializado que podemos llamar "**balanceador**". En la siguiente figura podemos ver un esquema en el cual partimos de un componente A, el cual recibe un número N de peticiones. El proceso de escalado horizontal es incorporar un componente de la misma naturaleza de A, y entre ellos se distribuirán las peticiones que se ejecutarán de manera colaborativa.



Fuente: Elemento balanceador para escalado horizontal. Extraída de Mora Pérez (2012).

2. Distribución de los datos:

Según Mora Pérez (2012), podemos asegurar que una plataforma es escalable horizontalmente si sus componentes o nodos siguen una arquitectura ***Shared Nothing***, donde cada nodo tiene sus propios recursos no compartidos, su propia memoria, almacenamiento e interfaces de entrada/salida independientes y se **comunica y sincroniza** con otros nodos a través de una red de comunicación. En la siguiente figura se compara la arquitectura de memoria (disco) compartida y la de nada compartido (*Shared Nothing*).



Fuente: Shared Nothing Architecture Diagram. Extraída de Simic (2021).

Para que el escalamiento sea posible, es necesaria la **distribución** de los datos entre los nodos del sistema.

3. Técnicas de distribución de los datos:

Los sistemas distribuidos usan principalmente las siguientes técnicas para la distribución de los datos:

- **Fragmentación o *sharding*** es el particionamiento de los datos en porciones más pequeñas para distribuirlos entre varios nodos del sistema; la reagrupación de las particiones representa la base de datos completa. Entre los tipos de fragmentación tenemos:

  - **Fragmentación horizontal**: el particionamiento se hace por registros o filas.
  - **Fragmentación vertical**: el particionamiento se hace por atributos o columnas.
  - **Mixta**: se combinan las dos anteriores.

Original Table

CUSTOMER ID	FIRST NAME	LAST NAME	CITY
1	Alice	Anderson	Austin
2	Bob	Best	Boston
3	Carrie	Conway	Chicago
4	David	Doe	Denver

Vertical Shards

VS1

CUSTOMER ID	FIRST NAME	LAST NAME
1	Alice	Anderson
2	Bob	Best
3	Carrie	Conway
4	David	Doe

VS2

CUSTOMER ID	CITY
1	Austin
2	Boston
3	Chicago
4	Denver

Horizontal Shards

HS1

CUSTOMER ID	FIRST NAME	LAST NAME	CITY
1	Alice	Anderson	Austin
2	Bob	Best	Boston

HS2

CUSTOMER ID	FIRST NAME	LAST NAME	CITY
3	Carrie	Conway	Chicago
4	David	Doe	Denver

Fuente: Sharding. Extraída de Hazelcast (s.f.).

El objetivo es que las aplicaciones hagan uso solo de los fragmentos que necesitan para responder a los requerimientos de los usuarios.

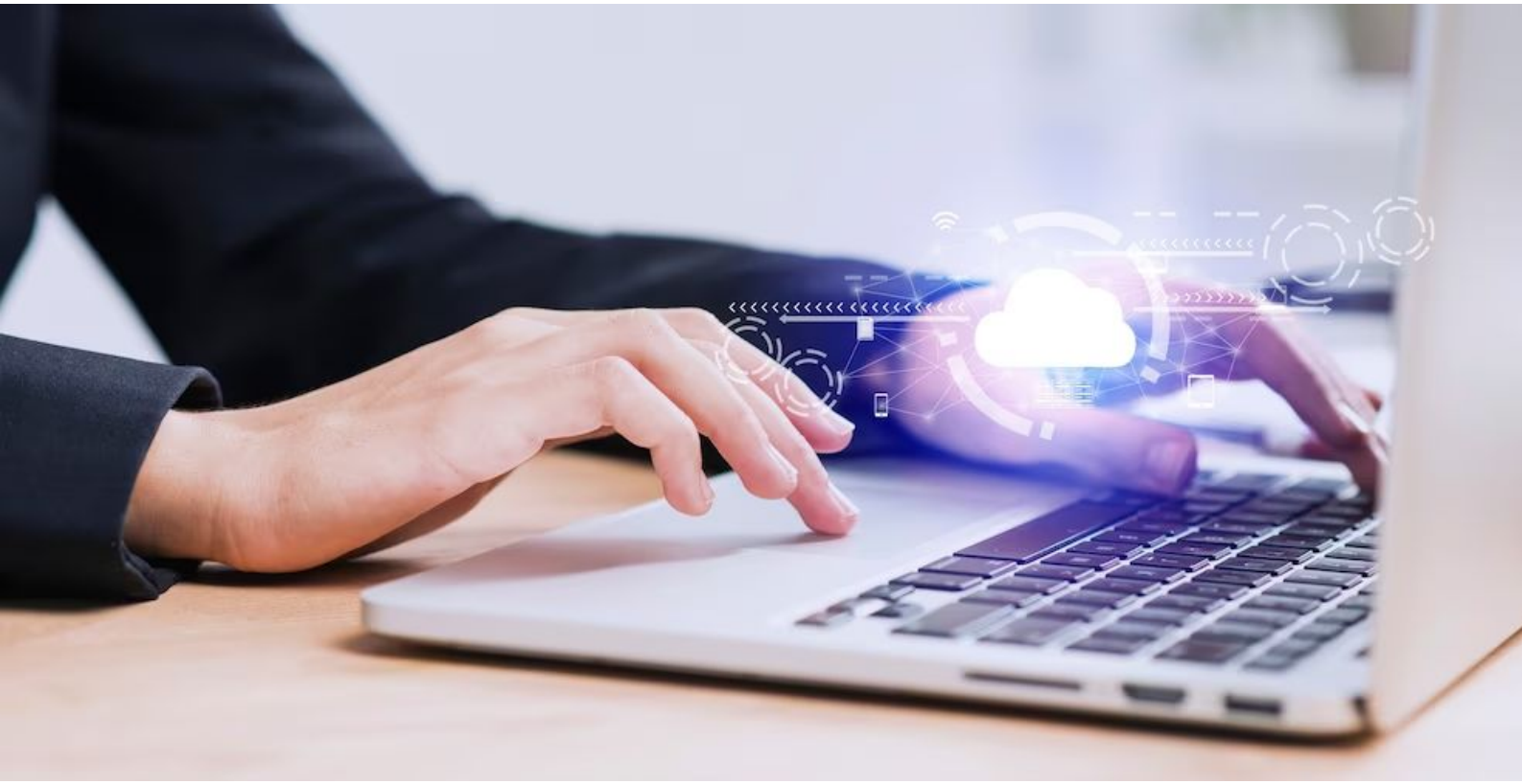
Entre los beneficios de la fragmentación se tienen:

- **Mayor concurrencia**: ya que los usuarios que requieran datos que están en diferentes fragmentos no competirán por el mismo recurso.
- **Paralelización de consultas**: si una consulta u operación requiere de distintos fragmentos, se puede operar sobre cada uno de manera paralela y luego procesar los diferentes resultados para dar una respuesta.
- **Distribución de carga**: por estar los fragmentos en diferentes nodos, las peticiones no irán a un solo nodo.

- **Replicación**: es el proceso de copiar y mantener actualizados los datos en varios nodos del sistema para que tengamos varias copias de los mismos datos disponibles en diferentes servidores (Sharma, 2021). En este proceso los lectores pueden leer cualquier copia y una escritura debe ser replicada en todas ellas.

Entre **los beneficios de la replicación** se tienen:

- **Mejora de la disponibilidad y accesibilidad de los datos:** si n copias accesibles fallan, aún se puede acceder a la copia n+1.
- **Mejora del rendimiento:** si hay más lecturas que escrituras, ya que los lectores pueden leer cualquier copia, preferiblemente la más cercana, por lo que los tiempos de respuesta disminuyen.
- **Recuperación de fallos:** al recuperarse un nodo puede acceder a los datos desde alguna de las réplicas activas.
- **Reducción de carga:** las operaciones sobre los datos se pueden repartir entre las diferentes réplicas.



- **Costo:** ya que requiere el uso de mayores recursos de almacenamiento.
- **Consistencia:** modificar una copia la hace diferente al resto.
- **Sobrecarga:** para garantizar la consistencia, las modificaciones se deben hacer en todas las copias. Además, se debe garantizar el manejo concurrente de cada una de ellas por los escritores. También puede darse el caso de que se consuma más ancho de banda de la red para mantener todas las réplicas actualizadas.
- **Fiabilidad:** se debe garantizar la seguridad de todas las copias almacenadas.

Entre los **tipos de replicación** se tienen:

- Según la cantidad de datos replicados:

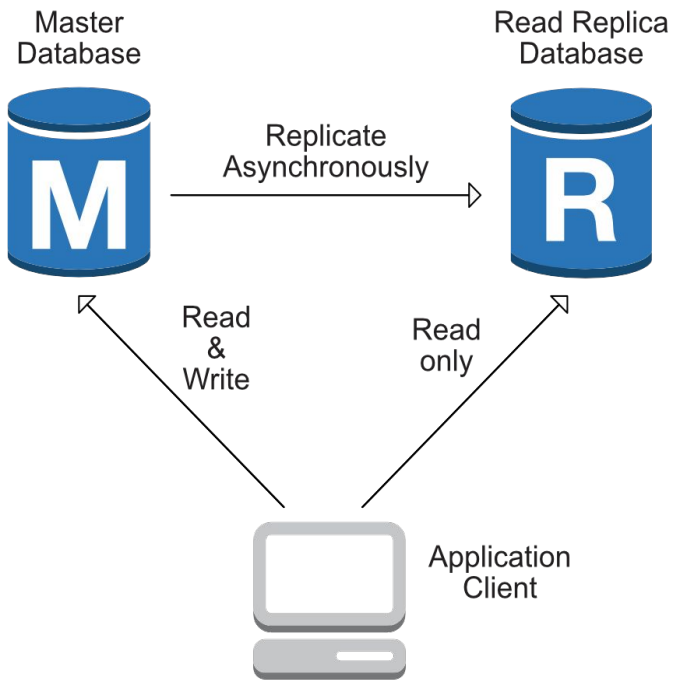
  - **Replicación total**: se replica la totalidad de los datos (la base de datos completa).
  - **Replicación parcial**: se replica solo una porción de los datos. Se replican algunos fragmentos de la base de datos.

- Según el momento:

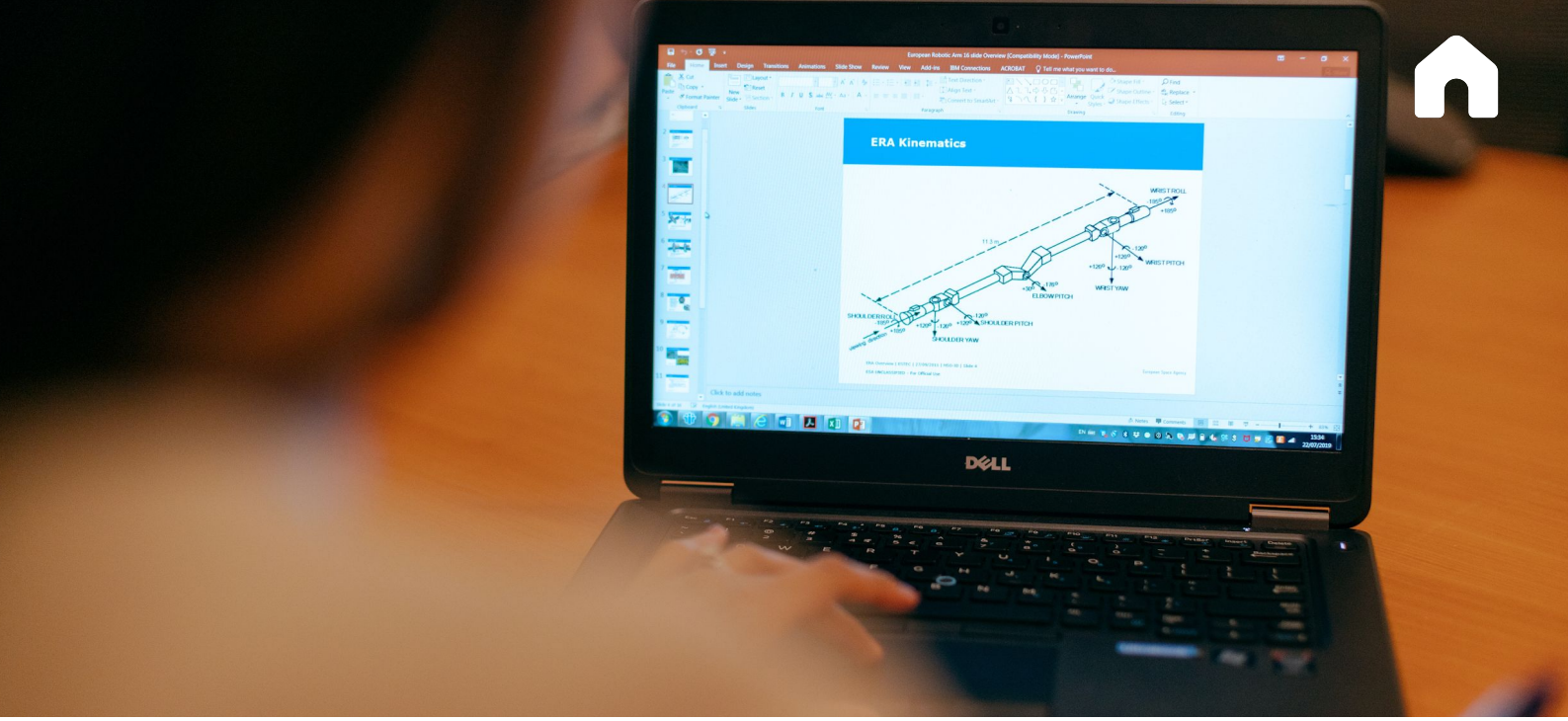
  - **Síncrona**: las copias se actualizan inmediatamente en el momento de actualizar los datos de origen (consistencia estricta).
  - **Asíncrona**: las copias se actualizan después de que los datos de origen hayan sido modificados (consistencia eventual).

- Según el modelo de replicación:

  - **Maestro/esclavo**: la réplica maestra o primaria recibe peticiones de lectura y escritura y es la que se encarga de mantener actualizadas a las réplicas esclavas o secundarias, las cuales solo pueden atender peticiones de lectura.

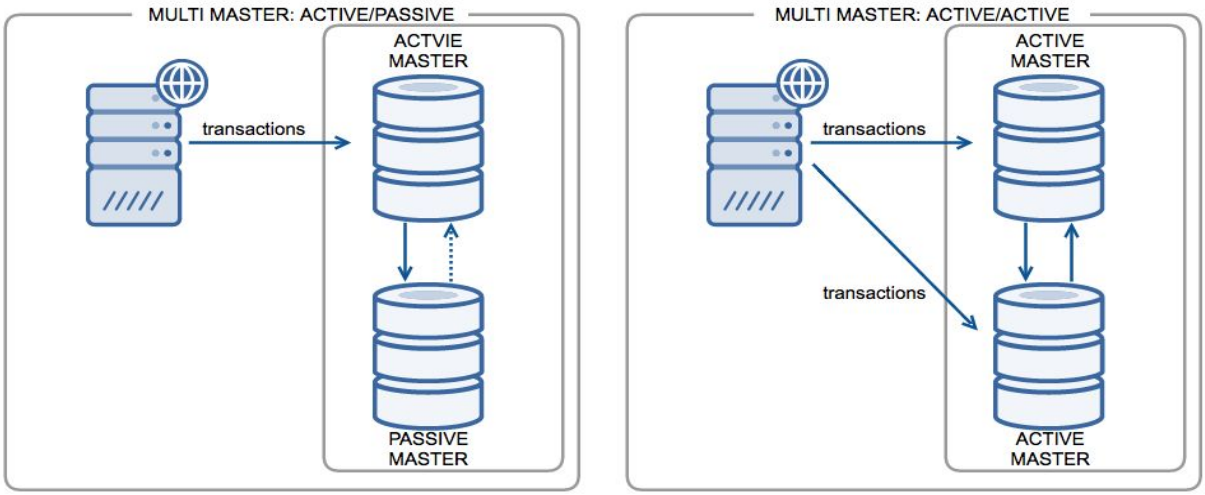


Fuente: Database Connections. Extraída de Iqmal (2021) .



Como **desventaja** primordial, la réplica principal representa un único punto de falla al momento de requerir actualizaciones en la base de datos.

- **Múltiples maestros**: cualquier nodo del conjunto de réplicas puede recibir peticiones de escrituras y lecturas, por lo tanto hay más de un nodo responsable de mantener la consistencia de las réplicas. Con esto se elimina el único punto de falla, pero se generan problemas de colisión si se ingresan dos registros distintos con la misma clave en diferentes réplicas. Las topologías **multimaestro** se pueden configurar para tener una configuración activa/pasiva, donde solo se puede escribir en un nodo y el otro nodo es un modo de espera activo. Luego está la configuración activo/activo, donde se puede escribir en ambos nodos



Fuente: Multi Master. Extraída de Several 9s

Los sistemas distribuidos, sus niveles de transparencia y sus arquitecturas son sistemas que permiten el escalamiento horizontal a través de las técnicas de distribución de datos, las cuales hacen posible el desarrollo de soluciones de ***big data***, ya que este escalamiento permite el poder procesar grandes volúmenes de datos cuyas velocidades dependen del rendimiento del sistema.

Blancarte, O. (2021). *Arquitectura Peer to Peer (P2P).* Reactive Programming. https://reactiveprogramming.io/blog/es/estilos-arquitectonicos/p2p

Liu, M. (2004). *Computación Distribuida Fundamentos y Aplicaciones.* Pearson Education S.A.

Mora Pérez, J. (2012). *Capacity Planning IT.* Autoedición.

Sharma, M. (2021). *MongoDB Complete Guide.* BPB Publications.

Tanenbaum, A. y Steen, M. (2008). *Sistemas Distribuidos. Principios y Paradigmas* Pearson Educación.

Referencias de las imágenes

Blancarte, O. (2021). Arquitectura P2P [Imagen]. Disponible en: *Arquitectura Peer to Peer (P2P).* Reactive Programming. https://reactiveprogramming.io/blog/es/estilos-arquitectonicos/p2p

Darvish, A. (2020). Interacción solicitud–respuesta de la arquitectura cliente-servidor [Imagen]. Disponible en: https://darvishdarab.github.io/cs421_f20/docs/readings/client_server/

Hazelcast (s.f.). Sharding [Imagen]. Disponible en: https://hazelcast.com/glossary/sharding/

Iqmal, H. (2021). Database Connections [Imagen]. Disponible en: https://hafiqiqmal93.medium.com/read-and-write-database-connections-in-laravel-a6f2e9b09b

ISO (1995). Tabla 1. Distintas formas de transparencia aplicables en un sistema distribuido [Imagen].

Mora Pérez, J. (2012). Elemento balanceador para escalado horizontal [Imagen]. Disponible en: *Capacity Planning IT: una aproximación práctica.*

Nunez, J. (2020). Arquitectura cliente-servidor [Imagen]. Disponible en: https://medium.com/@joshjnunez09/the-client-server-relationship-9ac90fadb3d2

Several 9s (2016). Multi Master [Imagen]. Disponible en: https://severalnines.com/blog/difference-between-mysql-multi-master-and-multi-source-replication/

Simic, S. (2021). Shared Nothing Architecture Diagram [Imagen]. Disponible en: https://phoenixnap.com/kb/shared-nothing-architecture

Singh, A. (2017). Arquitectura de 3 capas [Imagen]. Disponible en: https://msatechnosoft.in/blog/types-of-client-server-architecture/

UCAB UNIVERSIDAD CATÓLICA ANDRÉS BELLO

Has culminado la revisión del tema

Sistemas distribuidos, escalabilidad y distribución de los datos. por Mercy Ospina- UCAB, está bajo una Licencia Creative Commons Atribución-NoComercial-CompartirIgual 4.0 Internacional.